Typing linear algebra: A biproduct-oriented approach

Hugo Daniel Macedo, José N. Oliveira

Typing Linear Algebra: A Biproduct-oriented Approach

Hugo Daniel Macedo[a,1,*], José Nuno Oliveira[a]

[a]*HASLAB - High Assurance Software Laboratory*
Universidade do Minho, Braga, Portugal

Abstract

Interested in formalizing the generation of fast running code for linear algebra applications, the authors show how an index-free, calculational approach to matrix algebra can be developed by regarding matrices as morphisms of a category with biproducts. This shifts the traditional view of matrices as indexed structures to a type-level perspective analogous to that of the pointfree algebra of programming. The derivation of fusion, cancellation and abide laws from the biproduct equations makes it easy to calculate algorithms implementing matrix multiplication, the central operation of matrix algebra, ranging from its divide-and-conquer version to its vectorization implementation.

From errant attempts to learn how particular products and coproducts emerge from biproducts, not only blocked matrix algebra is rediscovered but also a way of extending other operations (e.g. Gaussian elimination) blockwise, in a calculational style, is found.

The prospect of building biproduct-based type checkers for computer algebra systems such as MATLAB™ is also considered.

Keywords: Linear algebra, categories of matrices, algebra of programming

> *"Using matrix notation such a set of simultaneous equations takes the form $A \cdot x = b$ where x is the vector of unknown values, A is the matrix of coefficients and b is the vector of values on the right side of the equation. In this way a set of equations has been reduced to a single equation. This is a tremendous improvement in concision that does not incur any loss of precision!"*
>
> Roland Backhouse [1]

1. Introduction

In a recent article [2], David Parnas questions the traditional use of formal methods in software development, which he regards unfit for the software industry. At the core of Parnas objections lies the contrast between the current ad-hoc (re)invention of cumbersome mathematical notation, often a burden to use, and elegant (thus useful) concepts which are neglected, often for cultural or (lack of) background reasons.

*Corresponding author.

[1]Partially supported by the *Fundação para a Ciência e a Tecnologia*, Portugal, under grant number SFRH/BD/33235/2007

The question is: what is it that tells "good" and "bad" methods apart? As Parnas writes, *there is a disturbing gap between software development and traditional engineering disciplines*. In such disciplines one finds a successful, well-established mathematical background essentially made of calculus, vector spaces, linear algebra and probability theory. This raises another question: can one hope to share such a successful tradition in the computing field, or is this definitely a different kind of science, hostage of formal logics and set theory?

There are signs of change in such direction already, as interest in the application of linear algebra techniques to computing seems to be growing, driven by disparate research interests briefly reviewed below.

Gunther Schmidt, for instance, makes extensive use of matrix notation, concepts and operations in his recent book on relational mathematics [3]. This pays tribute to binary relations being just Boolean matrices. Of historical relevance, explained in [4], is the fact of one of the first known definitions of relational composition, due to Charles Peirce (1839-1914), being essentially what we understand today as matrix multiplication.

In the area of process semantics, Bloom *et al* [5] have developed a categorical, *machines as matrices* approach to concurrency [2]; Trčka [7] presents a unifying matrix approach to the notions of strong, weak and branching bisimulation ranging from labeled transition systems to Markov reward chains; and Kleene coalgebra is going quantitative [8].

The "quantum inspiration" is also pushing computing towards linear algebra foundations. Focussing on quantum programming and semantics of probabilistic programs, Sernadas *et al* [9] adopt linear algebra techniques by regarding probabilistic programs as linear transformations over suitable vector spaces. Natural language semantics, too, is going vectorial, as nicely captured by the aphorism *nouns are vectors, adjectives are matrices* [10]. In this field of "quantum linguistics", Coecke *et al* [11] have developed a compositional model of meaning in which the grammatical structure of sentences is expressed in the category of finite dimensional vector spaces. Unrelated to quantum linguistics but related to knowledge discovery, the authors of the current paper show in [12] how to implement data mining operations solely based on linear algebra operations. And more examples of the adoption of linear algebra background in computing could be mentioned.

2. Typing Linear Algebra

One R&D field whose core lies in linear algebra (LA) is the automatic generation of fast running code for LA applications running on parallel architectures [13, 14, 15, 16]. The sophisticated techniques developed in this direction of research call for matrix multiplication as kernel operator, whereby matrices are viewed and transformed in an index-free way [16].

[2] Work in this vein can be traced much earlier, back to Conway's work on *regular algebras* [6] and regular algebras of matrices, so elegantly presented in textbook [1, Chap. 10] where the opening quotation of the current paper is taken from.

Interestingly, the successful language SPL [14] used in generating automatic parallel code has been created envisaging the same principles as advocated by the purist computer scientist: index-free abstraction and composition (multiplication) as a kernel way of connecting objects of interest (matrices, programs, etc).

There are several domain specific languages (DSLs) bearing such purpose in mind [14, 15, 16]. However, they arise as programming dialects with poor type checking. Of popular use and suffering from the same weakness one finds the widespread MATLAB [3] library of matrix operations, in which users have to keep track of dimensions all the way through and raise exceptions wherever "expressions don't fit with each other". This hinders effective use of such languages and libraries, calling for a "type structure" in linear algebra systems similar to that underlying modern functional programming languages such as Haskell, for instance [17].

It so happens that, in the same way function composition is the kernel operation of functional programming, leading to the *algebra of programming* [18], so does matrix multiplication once matrices are viewed and transformed in an index-free way. Therefore, rather than interpreting the product AB of matrices A and B as an algorithm for computing a new matrix C out of A and B, and trying to build and explain matrix algebra systems out of such an algorithm, one wishes to abstract from *how* the operation is carried out. Instead, the emphasis is put on its type structure, regarded as the pipeline $A \cdot B$ (to be read as "A after B"), as if A and B were functions

$$C \quad = \quad A \cdot B \tag{1}$$

or binary relations — the actual building block of the algebra of programming [18]. In this discipline, relations are viewed as (typed) composable arrows (morphisms) which can be combined in a number of ways, namely by joining or intersecting relations of the same type, reversing them (thus swapping their source and target types), and so on.

If relations, which are Boolean matrices, can be regarded as morphisms of a suitable mathematical framework [18, 19], why not regard arbitrary matrices in the same way? This matches with the categorical characterization of matrices, which can be traced back to Mac Lane [20], whereby matrices are regarded as arrows in a category whose objects are natural numbers (matrix dimensions):

$$A = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix}_{m \times n} \qquad\qquad m \xleftarrow{A} n \tag{2}$$

Such a category Mat_K of matrices over a field K merges categorical products and coproducts into a single construction termed *biproduct* [20]. Careful analysis of the biproduct axioms as a system of equations provides a rich *palette* of constructs for building matrices from smaller ones. In [21] we developed an approach to matrix blocked operation stemming from one particular solution to such equations, which in fact offers explicit operators for building block-wise matrices (row and column-wise) as defined by [5]. We also showed how divide-and-conquer algorithms for linear algebra arise from biproduct laws emerging from the underlying categorial basis.

[3] MATLAB ™ is a trademark of The MathWorks ®.

In the current paper we elaborate on [21] and show how biproduct-orientation leads into a simple, polymorphic type system for linear algebra. In the same way the categorial approach to functional programming — types-as-objects, functions-as-morphisms, etc [18] — leads into a widely acclaimed type-system, so one expects categories of matrices to offer a basis for typing linear algebra, as will be shown in this paper. Resistance to adopting such a categorial, but simple type system entails the need for more elaborate type mechanisms such as eg. *dependent types* [22] [4].

The paper includes three illustrations of biproduct-orientation: the implementation of matrix-matrix multiplication (MMM), a blocked version of the Gauss-Jordan elimination algorithm and a thorough study of vectorization, required in mapping matrices into computers' linear storage. Altogether, the paper gives the details of a constructive approach to matrix algebra operations leading to elegant, index-free proofs of the corresponding algorithms.

Structure of the paper. The remainder of this paper is structured as follows. Section 3 introduces the reader to categories of matrices and biproducts. Section 4 finds solutions to the biproduct equations, in particular those which explain blocked-matrix operations. Sections 5 and 6 develop a calculational approach to blocked linear algebra and present an application — that of calculating the nested-loop implementation of MMM. Section 7 shows how to develop biproduct algebra for applications, illustrated by the synthesis of a blocked-version of Gauss-Jordan elimination. Sections 8 and 9 show how the algebra of matrix vectorization emerges from a self-adjunction in the category of matrices whose unit and counit are expressed in terms of the underlying biproduct. Section 10 shows how to refine linear algebra operators once matrices are represented by vectors.

The remaining sections review related work and conclude, giving pointers for future research.

3. The Category of Matrices $\mathbf{Mat}_K$

Matrices are mathematical objects that can be traced back to ancient times, documented as early as 200 BC [23]. The word "matrix" was introduced in the western culture much later, in the 1840's, by the mathematician James Sylvester (1814-1897) when both matrix theory and linear algebra emerged.

The traditional way of viewing matrices as rectangular tables (2) of elements or entries (the "container view") which in turn are other mathematical objects such as e.g. complex numbers (in general: inhabitants of the field K which underlies Mat_K), encompasses as special cases one column and one line matrices, referred to as column (resp. row) *vectors*, that is, matrices of shapes

$$v = \begin{bmatrix} v_1 \\ \vdots \\ v_m \end{bmatrix} \qquad \text{and} \qquad w = \begin{bmatrix} w_1 & \dots & w_n \end{bmatrix}$$

[4] In fact, typing matrix operators provides a popular illustration of dependent types [22].

What is a matrix?. The standard answer to this question is to regard matrix A (2) as a computation unit, or transformation, which commits itself to producing a (column) vector of size m provided it is supplied with a (column) vector of size n. How is such output produced? Let us abstract from this at this stage and look at diagram

$$m \xleftarrow{A} n \xleftarrow{v} 1 \qquad m \xleftarrow{w} 1$$

arising from depicting the situation above in arrow notation. This suggests a pictorial representation of the product of matrix $A_{m\times n}$ and matrix $B_{n\times q}$, yielding a new matrix $C = (AB)_{m\times q}$ with dimensions $m \times q$, as follows,

$$m \xleftarrow{A} n \xleftarrow{B} q \qquad m \xleftarrow{C=A\cdot B} q \tag{3}$$

which automatically "type-checks" the construction: the "target" of $n \xleftarrow{B} q$ simply matches the "source" of $m \xleftarrow{A} n$ yielding a matrix whose type $m \longleftarrow q$ is the composition of the given types.

Having defined matrices as composable arrows in a category, we need to define its identities [20]: for every object n, there must be an arrow of type $n \longleftarrow n$ which is the unit of composition. This is nothing but the identity matrix of size n, which will be denoted by $n \xleftarrow{id_n} n$ or $n \xleftarrow{1} n$, indistinguishably. Therefore, for every matrix $m \xleftarrow{A} n$, equalities

$$id_m \cdot A \;=\; A \;=\; A \cdot id_n \qquad\qquad \begin{array}{ccc} n & \xleftarrow{id_n} & n \\ {\scriptstyle A}\downarrow & \swarrow_{A} & \downarrow{\scriptstyle A} \\ m & \xleftarrow[id_m]{} & m \end{array} \tag{4}$$

hold. (Subscripts m and n can be omitted wherever the underlying diagrams are assumed.)

Transposed matrices. One of the kernel operations of linear algebra is *transposition*, whereby a given matrix changes shape by turning its rows into columns and vice-versa. Type-wise, this means converting an arrow $n \xleftarrow{A} m$ into an arrow $m \xleftarrow{A^{\mathsf{T}}} n$, that is, source and target types (dimensions) switch over. By analogy with relation algebra, where a similar operation is termed *converse* and denoted A°, we will use this notation instead of A^{T} and will say "A converse" wherever reading A°. Index-wise, we have, for A as in (2):

$$A^{\circ} = \begin{bmatrix} a_{11} & \dots & a_{m1} \\ \vdots & \ddots & \vdots \\ a_{1n} & \dots & a_{mn} \end{bmatrix} \qquad\qquad m \xleftarrow{A^{\circ}} n$$

Instead of telling how transposition is carried out index-wise, again we prefer to stress on (index-free) properties of this operation such as, among others, idempotence and contravariance:

$$(A^\circ)^\circ = A \quad (5)$$

$$(A \cdot B)^\circ = B^\circ \cdot A^\circ \quad (6)$$

Bilinearity. Given two matrices of the same type $m \xleftarrow{A,B} n$ (i.e., in the same homset of Mat_K) it makes sense to add them up index-wise, leading to matrix $A + B$ where symbol + promotes the underlying element-level additive operator to matrix-level. Likewise, additive unit element 0 is promoted to matrix 0 wholy filled with 0s, the unit of matrix addition and zero of matrix composition:

$$A + 0 = A = 0 + A \quad (7)$$

$$A \cdot 0 = 0 = 0 \cdot A \quad (8)$$

In fact, matrices form an *Abelian category*: each homset in the category forms an additive Abelian (i.e. commutative) group with respect to which composition is bilinear:

$$A \cdot (B + C) = A \cdot B + A \cdot C \quad (9)$$

$$(B + C) \cdot A = B \cdot A + C \cdot A \quad (10)$$

Polynomial expressions (such as in the properties above) denoting matrices built up in an index-free way from addition and composition play a major role in matrix algebra. This can be appreciated in the explanation of the very important concept of a *biproduct* [20, 24] which follows.

Biproducts. In an Abelian category, a *biproduct* diagram for the objects m, n is a diagram of shape

$$m \underset{i_1}{\overset{\pi_1}{\leftrightarrows}} r \underset{i_2}{\overset{\pi_2}{\leftrightarrows}} n$$

whose arrows π_1, π_2, i_1, i_2 satisfy the identities which follow:

$$\pi_1 \cdot i_1 = id_m \quad (11)$$

$$\pi_2 \cdot i_2 = id_n \quad (12)$$

$$i_1 \cdot \pi_1 + i_2 \cdot \pi_2 = id_r \quad (13)$$

Morphisms π_i and i_i are termed *projections* and *injections*, respectively. From the underlying arithmetics one easily derives the following orthogonality properties (details in the appendix):

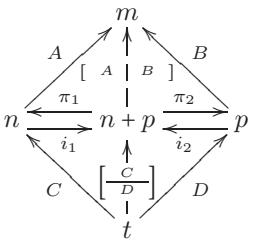
$$\pi_1 \cdot i_2 = 0 \quad (14)$$

$$\pi_2 \cdot i_1 = 0 \quad (15)$$

One wonders: how do biproducts relate to products and co-products in the category? The answer in Mac Lane's [20] words is as follows:

Theorem 2: Two objects a and b in Abelian category A have a product in A iff they have a biproduct in A. Specifically, given a biproduct diagram, the object r with the projections π_1 and π_2 is a product of m and n, while, dually, r with i_1 and i_2 is a coproduct. In particular, two objects m and n have a product in A if and only if they have a coproduct in A.

The diagram and definitions below depict how products and coproducts arise from biproducts (the product diagram is in the lower half; the upper half is the coproduct one):



$$\left[\begin{array}{c|c} A & B \end{array}\right] = A \cdot \pi_1 + B \cdot \pi_2 \quad (16)$$

$$\left[\frac{C}{D}\right] = i_1 \cdot C + i_2 \cdot D \quad (17)$$

By analogy with the algebra of programming [18], expressions $\left[\begin{array}{c|c} A & B \end{array}\right]$ and $\left[\frac{C}{D}\right]$ will be read "A junc B" and "C split D", respectively. What is the intuition behind these combinators, which come out of the blue in texts such as e.g. [5]? Let us start by a simple illustration, for $m = n = 2$, $p = 1$, $A = \left[\begin{smallmatrix} 1 & 2 \\ 4 & 5 \end{smallmatrix}\right]$, $B = \left[\begin{smallmatrix} 3 \\ 6 \end{smallmatrix}\right]$, $\pi_1 = \left[\begin{smallmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{smallmatrix}\right]$ and $\pi_2 = \left[\begin{smallmatrix} 0 & 0 & 1 \end{smallmatrix}\right]$. Then (16) instantiates as follows:

$$\begin{array}{ll} & \left[\begin{array}{c|c} A & B \end{array}\right] = A \cdot \pi_1 + B \cdot \pi_2 \\ = & \quad \{ \text{ instantiation } \} \\ & \left[\begin{array}{c|c} \begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} & \begin{bmatrix} 3 \\ 6 \end{bmatrix} \end{array}\right] = \begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} + \begin{bmatrix} 3 \\ 6 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \\ = & \quad \{ \text{ composition (3) } \} \\ & \left[\begin{array}{c|c} \begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} & \begin{bmatrix} 3 \\ 6 \end{bmatrix} \end{array}\right] = \begin{bmatrix} 1 & 2 & 0 \\ 4 & 5 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 3 \\ 0 & 0 & 6 \end{bmatrix} \\ = & \quad \{ \text{ matrix addition (7) } \} \\ & \left[\begin{array}{c|c} \begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} & \begin{bmatrix} 3 \\ 6 \end{bmatrix} \end{array}\right] = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \end{array}$$

A similar exercise would illustrate the split combinator (consider eg. transposing all arrows).

Expressed in terms of definitions (16) and (17), axiom (13) rewrites to both

$$\left[\begin{array}{c|c} i_1 & i_2 \end{array}\right] = id \quad (18)$$

$$\left[\frac{\pi_1}{\pi_2}\right] = id \quad (19)$$

somehow suggesting that the two injections and the two projections "decompose" the

identity matrix. On the other hand, each of (18,19) has the shape of a *reflection* corollary [18] of some universal property. Below we derive such a property for $[\ A \mid B\ ]$,

$$X = [\ A \mid B\ ] \quad \Leftrightarrow \quad \left\{ \begin{array}{l} X \cdot i_1 = A \\ X \cdot i_2 = B \end{array} \right. \tag{20}$$

from the underlying biproduct equations, by two-way implication:

$$\begin{array}{ll} & X = [\ A \mid B\ ] \\ \Leftrightarrow & \quad \{\ \text{identity (4) ; (16)}\ \} \\ & X \cdot id = A \cdot \pi_1 + B \cdot \pi_2 \\ \Leftrightarrow & \quad \{\ (13)\ \} \\ & X \cdot (i_1 \cdot \pi_1 + i_2 \cdot \pi_2) = A \cdot \pi_1 + B \cdot \pi_2 \\ \Leftrightarrow & \quad \{\ \text{bilinearity (9)}\ \} \\ & X \cdot i_1 \cdot \pi_1 + X \cdot i_2 \cdot \pi_2 = A \cdot \pi_1 + B \cdot \pi_2 \\ \Rightarrow & \quad \{\ \text{Leibniz (twice)}\ \} \\ & \left\{ \begin{array}{l} (X \cdot i_1 \cdot \pi_1 + X \cdot i_2 \cdot \pi_2) \cdot i_1 = (A \cdot \pi_1 + B \cdot \pi_2) \cdot i_1 \\ (X \cdot i_1 \cdot \pi_1 + X \cdot i_2 \cdot \pi_2) \cdot i_2 = (A \cdot \pi_1 + B \cdot \pi_2) \cdot i_2 \end{array} \right. \\ \Leftrightarrow & \quad \{\ \text{bilinearity (10) ; biproduct (11,12) ; orthogonality (15)}\ \} \\ & \left\{ \begin{array}{l} X \cdot i_1 + X \cdot i_2 \cdot 0 = A + B \cdot 0 \\ X \cdot i_1 \cdot 0 + X \cdot i_2 = A \cdot 0 + B \end{array} \right. \\ \Leftrightarrow & \quad \{\ \text{trivial}\ \} \\ & \left\{ \begin{array}{l} X \cdot i_1 = A \\ X \cdot i_2 = B \end{array} \right. \\ \Rightarrow & \quad \{\ \text{Leibniz (twice)}\ \} \\ & \left\{ \begin{array}{l} X \cdot i_1 \cdot \pi_1 = A \cdot \pi_1 \\ X \cdot i_2 \cdot \pi_2 = B \cdot \pi_2 \end{array} \right. \\ \Rightarrow & \quad \{\ \text{Leibniz}\ \} \\ & X \cdot i_1 \cdot \pi_1 + X \cdot i_2 \cdot \pi_2 = A \cdot \pi_1 + B \cdot \pi_2 \\ \Leftrightarrow & \quad \{\ \text{as shown above}\ \} \\ & X = [\ A \mid B\ ] \end{array}$$

The derivation of the universal property of $\left[\frac{C}{D}\right]$,

$$X = \left[\frac{C}{D} \right] \quad \Leftrightarrow \quad \left\{ \begin{array}{l} \pi_1 \cdot X = C \\ \pi_2 \cdot X = D \end{array} \right. \tag{21}$$

is (dually) analogous.

Last but not least, we stress that injections and projections in a biproduct are unique. Thus, for instance,

$$A \cdot \left[\frac{C}{D}\right] = C \ \wedge \ B \cdot \left[\frac{C}{D}\right] = D \quad \Leftrightarrow \quad A = \pi_1 \wedge B = \pi_2 \tag{22}$$

holds [5].

Remarks concerning notation. Outfix notation such as that used in *splits* and *juncs* provides for unambiguous parsing of matrix algebra expressions. Concerning infix operators (such as eg. composition, +) and unary ones (eg. converse, and others to appear) the following conventions will be adopted for saving parentheses: (a) unary and prefix operators bind tighter than binary; (b) multiplicative binary operators bind tighter than additive ones; (c) matrix multiplication (composition) binds tighter than any other multiplicative operator (eg. Kronecker product, to appear later).

We will resort to MATLAB notation to illustrate the main constructions of the paper. For instance, *split* $\left[\frac{A}{B}\right]$ (resp. *junc* $[\ A \mid B\ ]$) is written as `[A ; B]` (resp. `[A B]`) in MATLAB. More elaborate constructs will be encoded in the form of MATLAB functions.

Parallel with relation algebra. Similar to matrix algebra, relation algebra [3, 18, 25] can also be explained in terms of biproducts once morphism addition (13) is interpreted as relational union, object union as disjoint union, i_1 and i_2 as the corresponding injections and π_1, π_2 their converses, respectively [6]. Relational product should not, however, be confused with the *fork* construct [26] in fork (relation) algebra, which involves pairing. (For this to become a product one has to restrict to functions.)

It is worth mentioning that the matrix approach to relations, as intensively stressed in [3], is not restricted to set-theoretic models of allegories. For instance, Winter [27] builds categories of matrices on top of categories of relations.

In the next section we show that the converse relationship (duality) between projections and injections is not a privilege of relation algebra: the most intuitive biproduct solution in the category of matrices also offers such a duality.

4. Chasing biproducts

Let us now address the intuition behind products and coproducts of matrices. This has mainly to do with the interpretation of projections π_1, π_2 and injections i_1, i_2 arising as solutions of biproduct equations (11,12,13). Concerning this, Mac Lane [20] laconically writes:

> *"In other words, the [biproduct] equations contain the familiar calculus of matrices."*

[5] Easy to check: from right to left, just let $X := \left[\frac{C}{D}\right]$ in (22) and simplify; in the opposite direction, let $C, D := A, B$ in (22) and note that $\left[\frac{A}{B}\right] = id$ due to *split* uniqueness.

[6] Note that orthogonality (14, 15) is granted by the disjoint union construction itself.

sol = Simplify[Solve[{pi1.i1 == I1, pi2.i2 == I1, i1.pi1 + i2.pi2 == I2}]]

Solve::svars : Equations may not give solutions for all "solve" variables.

$$\left\{\left\{i_{11} \to -\frac{\pi_{22}}{\pi_{12}\pi_{21}}, i_{12} \to \frac{1}{\pi_{12}}, i_{21} \to \frac{1}{\pi_{21}}, i_{22} \to 0, \pi_{11} \to 0\right\},\right.$$
$$\left\{i_{11} \to \frac{\pi_{22}}{-\pi_{12}\pi_{21} + \pi_{11}\pi_{22}}, i_{12} \to \frac{\pi_{21}}{\pi_{12}\pi_{21} - \pi_{11}\pi_{22}},\right.$$
$$\left.\left.i_{21} \to \frac{\pi_{12}}{\pi_{12}\pi_{21} - \pi_{11}\pi_{22}}, i_{22} \to \frac{\pi_{11}}{-\pi_{12}\pi_{21} + \pi_{11}\pi_{22}}\right\}\right\}$$

Figure 1: Fragment of Mathematica script

In what way? The answer to this question proves more interesting than it seems at first, because of the multiple solutions arising from a non-linear system of three equations (11,12,13) with four variables. In trying to exploit this freedom we became aware that each solution offers a particular way of putting matrices together via the corresponding "junc" and "split" combinators.

Our inspection of solutions started by reducing the "size" of the objects involved and experimenting with the smaller biproduct depicted below:

$$1 \underset{i_1}{\overset{\pi_1}{\leftrightarrows}} 1 + 1 \underset{i_2}{\overset{\pi_2}{\leftrightarrows}} 1$$

The "puzzle" in this case is more manageable,

$$\begin{cases} \pi_1 \cdot i_1 & = & [1] \\ \pi_2 \cdot i_2 & = & [1] \\ i_1 \cdot \pi_1 + i_2 \cdot \pi_2 & = & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \end{cases}$$

yet the set of solutions is not small. We used the Mathematica software [28] to solve this system by inputting the projections and injections as suitably typed matrices leading to a larger, non-linear system:

$$\begin{cases} \begin{bmatrix} \pi_{11} & \pi_{12} \end{bmatrix} \cdot \begin{bmatrix} i_{11} \\ i_{12} \end{bmatrix} & = & [1] \\ \begin{bmatrix} \pi_{21} & \pi_{22} \end{bmatrix} \cdot \begin{bmatrix} i_{21} \\ i_{22} \end{bmatrix} & = & [1] \\ \begin{bmatrix} i_{11} \\ i_{12} \end{bmatrix} \cdot \begin{bmatrix} \pi_{11} & \pi_{12} \end{bmatrix} + \begin{bmatrix} i_{21} \\ i_{22} \end{bmatrix} \cdot \begin{bmatrix} \pi_{21} & \pi_{22} \end{bmatrix} & = & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \end{cases}$$

This was solved using the standard Solve command obtaining the output presented in Figure 1, which offers several solutions. Among these we first picked the one which purports the most intuitive reading of the *junc* and *split* combinators — that of simply gluing matrices vertically and horizontally (respectively) with no further computation of matrix entries:

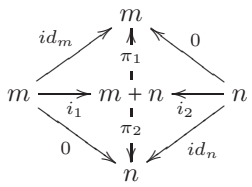
$$\pi_1 = \begin{bmatrix} 1 & 0 \end{bmatrix} \qquad \pi_2 = \begin{bmatrix} 0 & 1 \end{bmatrix}$$
$$i_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad i_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Interpreted in this way, $\left[\frac{A}{B}\right]$ (17) and $[\ A \mid B\ ]$ (16) are the block gluing matrix operators which one can find in [5]. Our choice of notation — A above B in the case of (17) and A besides B in the case of (16) reflects this semantics.

The obvious generalization of this solution to higher dimensions of the problem leads to the following matrices with identities of size m and n in the appropriate place, so as to properly typecheck [7]:

$$\begin{array}{lll} \pi_1 = m \xleftarrow{[\ id_m \mid 0\ ]} m+n & , & \pi_2 = n \xleftarrow{[\ 0 \mid id_n\ ]} m+n \\ i_1 = m+n \xleftarrow{\left[\frac{id_m}{0}\right]} m & , & i_2 = m+n \xleftarrow{\left[\frac{0}{id_m}\right]} n \end{array} \tag{23}$$

The following diagram pictures not only the construction of this biproduct but also the biproduct (11,12) and orthogonality (14, 15) equations — check the commuting triangles:



By inspection, one immediately infers the same duality found in relation algebra,

$$\pi_1^\circ = i_1 \quad , \quad \pi_2^\circ = i_2 \tag{24}$$

whereby *junc* (16) and *split* (17) become self dual:

$$\begin{array}{c} [\ R \mid S\ ]^\circ \\ = \quad \{\ (16)\ ;\ (6)\ \} \\ \pi_1^\circ \cdot R^\circ + \pi_2^\circ \cdot S^\circ \end{array}$$

[7] Projections π_1, π_2 (resp. injections i_1, i_2) are referred to as *gather* (resp. *scatter*) matrices in [29]. MATLAB's (untyped) notation for projection π_1 and injection i_1 in (23) is `eye(m,m+n)` and `eye(m+n,m)`, respectively. Consistently, `eye(n,n)` denotes id_n. Matrices π_2 and π_2 can be programmed using MATLAB's `eye` and `zeros` — see Listing 4 further on.

$$= \qquad \{\ (24)\ ;\ (17)\ \}$$

$$\left[\frac{R^\circ}{S^\circ}\right] \tag{25}$$

This particular solution to the biproduct equations captures what in the literature is meant by *blocked* matrix algebra, a generalization of the standard element-wise operations to sub-matrices, or blocks, leading to *divide-and-conquer* versions of the corresponding algorithms. The next section shows the exercise of deriving such laws, thanks to the algebra which emerges from the universal properties of the block-gluing matrix combinators *junc* (20) and *split* (21). We combine the standard terminology with that borrowed from the algebra of programming [18] to stress the synergy between blocked matrix algebra and relation algebra.

5. Blocked Linear Algebra — calculationally!

Further to reflection laws (18,19), the derivation of the following equalities from universal properties (20,21) is a standard exercise in (high) school algebra, where capital letters A, B, etc. denote suitably typed matrices (the types, i.e. dimensions, involved in each equality can be inferred by drawing the corresponding diagram):

- Two "fusion"-laws:

$$C \cdot \left[\ A \mid B\ \right] = \left[\ C \cdot A \mid C \cdot B\ \right] \tag{26}$$

$$\left[\frac{A}{B}\right] \cdot C = \left[\frac{A \cdot C}{B \cdot C}\right] \tag{27}$$

- Four "cancellation"-laws [8]:

$$\left[\ A \mid B\ \right] \cdot i_1 = A \quad , \quad \left[\ A \mid B\ \right] \cdot i_2 = B \tag{28}$$

$$\pi_1 \cdot \left[\frac{A}{B}\right] = A \quad , \quad \pi_2 \cdot \left[\frac{A}{B}\right] = B \tag{29}$$

- Three "abide"-laws [9]: the *junc/split* exchange law

$$\left[\frac{\left[\ A \mid B\ \right]}{\left[\ C \mid D\ \right]}\right] = \left[\ \left[\frac{A}{C}\right] \Big| \left[\frac{B}{D}\right]\ \right] = \left[\frac{A \mid B}{C \mid D}\right] \tag{30}$$

[8] Recall (22).

[9] Neologism "abide" (= "above and beside") was introduced by Richard Bird [30] as a generic name for algebraic laws in which two binary operators written in infix form change place between "above" and "beside", e.g.

$$\frac{a}{b} \times \frac{c}{d} = \frac{a \times c}{b \times d}$$

which tells the equivalence between row-major and column-major construction of matrices (thus the four entry *block* notation on the right), and two *blocked addition* laws:

$$\left[\begin{array}{c|c} A & B \end{array}\right] + \left[\begin{array}{c|c} C & D \end{array}\right] = \left[\begin{array}{c|c} A+C & B+D \end{array}\right] \quad (31)$$

$$\left[\frac{A}{B}\right] + \left[\frac{C}{D}\right] = \left[\frac{A+C}{B+D}\right] \quad (32)$$

- Two structural equality laws (over the same biproduct):

$$\left[\begin{array}{c|c} A & B \end{array}\right] = \left[\begin{array}{c|c} C & D \end{array}\right] \Leftrightarrow A = C \wedge B = D \quad (33)$$

$$\left[\frac{A}{B}\right] = \left[\frac{C}{D}\right] \Leftrightarrow A = C \wedge B = D \quad (34)$$

The laws above are more than enough for us to derive standard linear algebra rules and algorithms in a calculational way. As an example of their application we provide a simple proof of the rule which underlies *divide-and-conquer* matrix multiplication:

$$\left[\begin{array}{c|c} A & B \end{array}\right] \cdot \left[\frac{C}{D}\right] = A \cdot C + B \cdot D \quad (35)$$

We calculate:

$$\begin{array}{ll} & \left[\begin{array}{c|c} A & B \end{array}\right] \cdot \left[\frac{C}{D}\right] \\ = & \quad \{ \text{ (17) } \} \\ & \left[\begin{array}{c|c} A & B \end{array}\right] \cdot (i_1 \cdot C + i_2 \cdot D) \\ = & \quad \{ \text{ bilinearity (9) } \} \\ & \left[\begin{array}{c|c} A & B \end{array}\right] \cdot i_1 \cdot C + \left[\begin{array}{c|c} A & B \end{array}\right] \cdot i_2 \cdot D \\ = & \quad \{ \text{ +-cancellation (28) } \} \\ & A \cdot C + B \cdot D \end{array}$$

Listing 1 converts this law into the corresponding MATLAB algorithm for matrix multiplication.

As another example, let us show how standard block-wise matrix-matrix multiplication (MMM),

$$\left[\begin{array}{c|c} R & S \\ \hline T & U \end{array}\right] \cdot \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right] = \left[\begin{array}{c|c} R \cdot A + S \cdot C & R \cdot B + S \cdot D \\ \hline T \cdot A + U \cdot C & T \cdot B + U \cdot D \end{array}\right] \quad (36)$$

relies on *divide-and-conquer* (35):

$$
\begin{array}{ll}
 & \left[\left[\frac{R}{T}\right]\middle|\left[\frac{S}{U}\right]\right]\cdot\left[\left[\frac{A}{C}\right]\middle|\left[\frac{B}{D}\right]\right] \\
= & \quad \{\ \textit{junc}\text{-fusion (26)}\ \} \\
 & \left[\left[\left[\frac{R}{T}\right]\middle|\left[\frac{S}{U}\right]\right]\cdot\left[\frac{A}{C}\right]\middle|\left[\left[\frac{R}{T}\right]\middle|\left[\frac{S}{U}\right]\right]\cdot\left[\frac{B}{D}\right]\right] \\
= & \quad \{\ \text{divide and conquer (35) twice}\ \} \\
 & \left[\left[\frac{R}{T}\right]\cdot A+\left[\frac{S}{U}\right]\cdot C\ \middle|\ \left[\frac{R}{T}\right]\cdot B+\left[\frac{S}{U}\right]\cdot D\right] \\
= & \quad \{\ \text{split-fusion (26) four times}\ \} \\
 & \left[\left[\frac{R\cdot A}{T\cdot A}\right]+\left[\frac{S\cdot C}{U\cdot C}\right]\ \middle|\ \left[\frac{R\cdot B}{T\cdot B}\right]+\left[\frac{S\cdot D}{U\cdot D}\right]\right] \\
= & \quad \{\ \text{blocked addition (32) twice}\ \} \\
 & \left[\left[\frac{R\cdot A+S\cdot C}{T\cdot A+U\cdot C}\right]\ \middle|\ \left[\frac{R\cdot B+S\cdot D}{T\cdot B+U\cdot D}\right]\right] \\
= & \quad \{\ \text{the same in block notation (30)}\ \} \\
 & \left[\begin{array}{c|c} R\cdot A+S\cdot C & R\cdot B+S\cdot D \\ \hline T\cdot A+U\cdot C & T\cdot B+U\cdot D \end{array}\right]
\end{array}
$$

6. Calculating Triple Nested Loops

By putting together the universal factorization of matrices in terms of the *junc* and *split* combinators, one easily infers yet another such property handling four blocks at a time:

$$
X=\left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \quad \Leftrightarrow \quad \left\{\begin{array}{l} \pi_1\cdot X\cdot i_1=A_{11} \\ \pi_1\cdot X\cdot i_2=A_{12} \\ \pi_2\cdot X\cdot i_1=A_{21} \\ \pi_2\cdot X\cdot i_2=A_{22} \end{array}\right.
$$

Alternatively, one may generalize (16,17) to blocked notation

$$
\left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right]=i_1\cdot A_{11}\cdot\pi_1+i_1\cdot A_{12}\cdot\pi_2+i_2\cdot A_{21}\cdot\pi_1+i_2\cdot A_{22}\cdot\pi_2
$$

which rewrites to

$$
\left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right]=\begin{bmatrix} A_{11} & 0 \\ 0 & 0 \end{bmatrix}+\begin{bmatrix} 0 & A_{12} \\ 0 & 0 \end{bmatrix}+\begin{bmatrix} 0 & 0 \\ A_{21} & 0 \end{bmatrix}+\begin{bmatrix} 0 & 0 \\ 0 & A_{22} \end{bmatrix} \tag{37}
$$

once injections and projections are replaced by the biproduct solution of Section 4.

```
function R = MMM(X,Y)
   [k1, n] = size(Y);
   [m, k2] = size(X);
   if (k1 ~= k2)
      error('Dimensions must agree');
   else
      k = k1;
      R = zeros(n, m);
      if k1 == 1
         R = X * Y;
      else
         k1 = round(k / 2);
         A = X(:,1:k1); B = X(:,k1+1:k);
         C = Y(1:k1,:); D = Y(k1+1:k,:);
         R = MMM(A,C) + MMM(B,D);
      end
   end
end
```

Listing 1: Divide-and-conquer law (35) converted to MATLAB script for matrix-matrix multiplication. Blocks A, B in (35) are generated by partitioning argument matrix X column-wise and blocks C, D are obtained in a similar way from Y. The algorithm stops when both argument matrices degenerate into vectors ($k = 1$). There is no type checking, meaning that function MMM issues an error when the two **size** operations don't match — the number of columns (resp. lines) of X (resp. Y) must be the same.

Iterated Biproducts. It should be noted that biproducts generalize to finitely many arguments, leading to an n-ary generalization of the (binary) *junc*/*split* combinators. The following notation is adopted in generalizing (16,17):

$$\left[\begin{array}{c|c|c} A_1 & \ldots & A_p \end{array}\right] = \bigoplus_{1\leq j\leq p} A_j = \sum_{j=1}^{p} A_j \cdot \pi_j$$

$$\left[\begin{array}{c} A_1 \\ \hline \vdots \\ \hline A_m \end{array}\right] = \bigominus_{1\leq j\leq m} A_j = \sum_{j=1}^{m} i_j \cdot A_j$$

Note that all laws given so far generalize accordingly to n-ary *splits* and *juncs*. In particular, we have the following universal properties:

$$X = \bigoplus_{1\leq j\leq p} A_j \quad \Leftrightarrow \quad \bigwedge_{1\leq j\leq p} X \cdot i_j = A_j \tag{38}$$

$$X = \bigominus_{1\leq j\leq m} A_j \quad \Leftrightarrow \quad \bigwedge_{1\leq j\leq m} \pi_j \cdot X = A_j \tag{39}$$

The following rules expressing the block decomposition of a matrix A

$$A = \left[\begin{array}{c|c|c} A_1 & \ldots & A_p \end{array}\right] = \bigoplus_{1\leq j\leq p} A \cdot i_j = \sum_{j=1}^{p} A \cdot i_j \cdot \pi_j \tag{40}$$

$$A = \left[\begin{array}{c} A_1 \\ \hline \vdots \\ \hline A_m \end{array}\right] \quad = \quad \bigoplus_{1\leq j\leq m} \pi_j \cdot A = \sum_{j=1}^{m} i_j \cdot \pi_j \cdot A \tag{41}$$

arise from the iterated definitions by letting $X = A$ in the universal properties and substituting.

Further note that m, p can be chosen as large as possible, the limit taking place when blocks A_i become atomic. In this limit situation, a given matrix $m \xleftarrow{A} n$ is defined in terms of its elements A_{jk} as:

$$A = \left[\begin{array}{c|c|c} a_{11} & \cdots & a_{1n} \\ \hline \vdots & \ddots & \vdots \\ \hline a_{m1} & \cdots & a_{mn} \end{array}\right] \quad = \quad \sum_{\substack{1\leq j\leq m \\ 1\leq k\leq n}} i_j \cdot \pi_j \cdot A \cdot i_k \cdot \pi_k = \bigoplus_{\substack{1\leq j\leq m \\ 1\leq k\leq n}} \pi_j \cdot A \cdot i_k \tag{42}$$

where $\bigoplus_{\substack{1\leq j\leq m \\ 1\leq k\leq n}}$ abbreviates $\ominus_{1\leq j\leq m} \oplus_{1\leq k\leq n}$ — equivalent to $\oplus_{1\leq k\leq n} \ominus_{1\leq j\leq m}$ by the generalized exchange law (30).

Our final calculation shows how iterated biproducts "explain" the traditional for-loop implementation of MMM. Interestingly enough, such iterative implementation is shown to stem from generalized divide-and-conquer (35):

$$\begin{array}{lll} C & = & A \cdot B \\ & = & \quad \{ \text{ (42), (40) and (41) } \} \\ & & (\bigoplus_{1\leq j\leq m} \pi_j \cdot A) \cdot (\bigoplus_{1\leq k\leq n} B \cdot i_k) \\ & = & \quad \{ \text{ generalized split-fusion (27) } \} \\ & & \bigoplus_{1\leq j\leq m} (\pi_j \cdot A \cdot (\bigoplus_{1\leq k\leq n} B \cdot i_k)) \\ & = & \quad \{ \text{ generalized either-fusion (26) } \} \\ & & \bigoplus_{1\leq j\leq m} (\bigoplus_{1\leq k\leq n} \pi_j \cdot A \cdot B \cdot i_k) \\ & = & \quad \{ \text{ (40), (41) and generalized (27) and (26) } \} \\ & & \bigoplus_{1\leq j\leq m} (\bigoplus_{1\leq k\leq n} ((\bigoplus_{1\leq l\leq p} \pi_j \cdot A \cdot i_l) \cdot (\bigoplus_{1\leq l\leq p} \pi_l \cdot B \cdot i_k))) \\ & = & \quad \{ \text{ generalized divide-and-conquer (35) } \} \\ & & \bigoplus_{1\leq j\leq m} (\bigoplus_{1\leq k\leq n} (\sum_{1\leq l\leq p} \pi_j \cdot A \cdot i_l \cdot \pi_l \cdot B \cdot i_k)) \end{array}$$

As we can see in the derivation path, the choices for the representation of A and B impact on the derivation of the intended algorithm. Different choices will alter the order of the triple loop obtained. Proceeding to the loop inference will involve the

expansion of C and the normalization of the formula into sum-wise notation:

$$\bigoplus_{\substack{1\le k\le m\\ 1\le j\le n}} \pi_j\cdot C\cdot i_k = \bigominus_{1\le j\le m} \left(\bigoplus_{1\le k\le n} \left(\sum_{1\le l\le p} \pi_j\cdot A\cdot i_l\cdot \pi_l\cdot B\cdot i_k\right)\right)$$

$$\Leftrightarrow \qquad \{\ (42),\ (40)\ \text{and}\ (41)\ \}$$

$$\bigominus_{1\le j\le m} \left(\bigoplus_{1\le k\le n} \pi_j\cdot C\cdot i_k\right) = \bigominus_{1\le j\le m} \left(\bigoplus_{1\le k\le n} \left(\sum_{1\le l\le p} \pi_j\cdot A\cdot i_l\cdot \pi_l\cdot B\cdot i_k\right)\right)$$

At this point we rely on the universality of the *junc* and *split* constructs (38,39) to obtain from above the post-condition of the algorithm:

$$\bigwedge_{1\le j\le m} \left(\bigwedge_{1\le k\le n} \left(\pi_j\cdot C\cdot i_k = \sum_{1\le l\le p} \pi_j\cdot A\cdot i_l\cdot \pi_l\cdot B\cdot i_k\right)\right) \tag{43}$$

This predicate expresses an outer traversal indexed by j, an inner traversal indexed by k and what the expected result in each element of output matrix C is. Thus we reach three nested for-loops of two different kinds: the two outer-loops (corresponding to indices j, k) provide for *navigation*, while the inner loop performs an *accumulation* (thus the need for the initialization).

```
function C = NaiveMMM(A,B)

  [m, p1] = size(A);
  [p2, n] = size(B);

  if (p1 ~= p2)
      error('Dimensions must agree');
  else
   for j = 1:m
     for k = 1:n
       C(j,k) = 0;
       for l = 1:p1
         C(j,k) = C(j,k) + A(j,l) * B(l,k);
       end
     end
   end
  end

end
```

Listing 2: MATLAB encoding of naive triple `for`-loop implementation of MMM, corresponding to traversing the rows of A through j and the columns of B via k. This is a refinement of the calculated post-condition (43).

Different matrix memory mapping schemes give rise to the interchange of the j, k and l in the loops in Listing 2. (For a complete discussion of matrix partition possibilities see [31].) This is due to corresponding choices in the derivation granted by the generalized exchange law (30), among others.

Other variants of blocked MMM (36) such as e.g. Strassen's or Winograd's [32] rely mainly on the additive structure of Mat_K and thus don't pose new challenges.

7. Developing biproduct algebra for applications

For a mathematical concept to be effective it should blend expressiveness with calculation power, while providing a generic setting wherefrom practically relevant situations can be derived by instantiation. It should also scale up, in the sense of exhibiting an algebra making it easy to "build new from old".

We will see shortly that biproducts scale up in this manner. So, instead of chasing new solutions to the biproduct equations and checking which "chapters" of linear algebra [24] they are able to constructively explain, one may try and find rules which build new biproducts from existing ones so as to fit into particular situations in linear algebra.

Think of Gaussian elimination, for instance, whose main steps involve row-switching, row-multiplication and row-addition, and suppose one defines the following transformation t catering for the last two, for a given α:

$$t : (\ n \xleftarrow{\quad} n\ ) \times (\ n+n \xleftarrow{\quad} m\ ) \to (\ n+n \xleftarrow{\quad} m\ )$$
$$t(\alpha, \left[\frac{A}{B}\right]) \ = \ \left[\frac{A}{\alpha A + B}\right]$$

Thinking in terms of blocks A and B rather than rows is more general; in this setting, arrow $n \xleftarrow{\alpha} n$ means $n \xleftarrow{id} n$ with all 1s replaced by αs, and αA is $\alpha \cdot A$. Let us analyze transformation t in this setting, by using the blocked-matrix calculus in reverse order:

$$\begin{array}{lcl} t(\alpha, \left[\frac{A}{B}\right]) & = & \left[\frac{A}{\alpha \cdot A + B}\right] \\ & = & \quad \{\ \text{(36) in reverse order}\ \} \\ & & \left[\begin{array}{c|c} 1 & 0 \\ \hline \alpha & 1 \end{array}\right] \cdot \left[\frac{A}{B}\right] \\ & = & \quad \{\ \text{divide-and-conquer (35)}\ \} \\ & & \left[\frac{1}{\alpha}\right] \cdot A + \left[\frac{0}{1}\right] \cdot B \end{array}$$

It can be shown that the last expression, which has the same shape as (17), is in fact the *split* combinator generated by another biproduct,

$$\pi'_1 = \left[\begin{array}{cc} 1 & 0 \end{array}\right] \quad , \quad \pi'_2 = \left[\begin{array}{cc} -\alpha & 1 \end{array}\right]$$
$$i'_1 = \left[\begin{array}{c} 1 \\ \alpha \end{array}\right] \quad , \quad i'_2 = \left[\begin{array}{c} 0 \\ 1 \end{array}\right]$$

parametric on α. In summary, this biproduct, which extends the one studied earlier on (they coincide for $\alpha := 0$) provides a categorial interpretation of one of the steps of Gaussian elimination.

Biproducts in Mat_K are unique up to isomorphism due to universality of product and coproduct. *Splitting* π'_1 and π'_2 with the standard projections (23) is just another way to build *elementary matrices* [33] such as, for instance,

$$\left[\begin{array}{c} \pi'_1 \\ \hline \pi'_2 \end{array}\right] = i_1 \cdot \pi'_1 + i_2 \cdot \pi'_2 = \begin{bmatrix} 1 & 0 \\ -\alpha & 1 \end{bmatrix}$$

which are central to Gaussian elimination. In essence, this algorithm performs successive transformations of a given matrix into isomorphic ones via elementary matrices that witness the isomorphisms. Below we show that such elementary steps of Gaussian elimination scale up to blocks via suitable biproduct constructions. The first one generalizes row switching to block switching.

Theorem 1 (Swapping biproducts). *Let* $m \underset{i_1}{\overset{\pi_1}{\leftrightarrows}} r \underset{i_2}{\overset{\pi_2}{\leftrightarrows}} n$ *be a biproduct. Then swapping projections (resp. injections) with each other yields another biproduct.*

Proof: *Obvious, as (12) swaps with (11) and (13) stays the same, since addition is commutative.*

For instance, swapping the standard biproduct yields another biproduct (superscript s stands for *swap*) :

$$\begin{array}{ll} i_1^s = \left[\begin{array}{c} 0 \\ \hline 1 \end{array}\right] & i_2^s = \left[\begin{array}{c} 1 \\ \hline 0 \end{array}\right] \\ \pi_1^s = \left[\begin{array}{c|c} 0 & 1 \end{array}\right] & \pi_2^s = \left[\begin{array}{c|c} 1 & 0 \end{array}\right] \end{array} \tag{44}$$

Thus

$$\left[\begin{array}{c|c} A & B \end{array}\right]^s = \left[\begin{array}{c|c} B & A \end{array}\right] \tag{45}$$

$$\left[\begin{array}{c} A \\ \hline B \end{array}\right]^s = \left[\begin{array}{c} B \\ \hline A \end{array}\right] \tag{46}$$

Swapped biproduct (44) generalizes row-swapping to block-swapping, as the following example shows: the effect of swapping A with B in matrix $\left[\begin{smallmatrix} A \\ C \\ B \end{smallmatrix}\right]$ is obtained by representing it in *swap mode*:

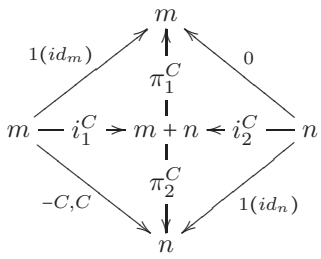
$$\left[\begin{array}{c} A \\ \hline \left[\begin{array}{c} C \\ \hline B \end{array}\right]^s \end{array}\right]^s = \left[\begin{array}{c} \left[\begin{array}{c} C \\ \hline B \end{array}\right]^s \\ \hline A \end{array}\right] = \left[\begin{array}{c} \left[\begin{array}{c} B \\ \hline C \end{array}\right] \\ \hline A \end{array}\right] = \left[\begin{array}{c} B \\ \hline C \\ \hline A \end{array}\right]$$

Next we want to show how to perform row-multiplication and addition at block-level. There is a biproduct for this, also evolving from the standard:

Theorem 2 (Self cancellable biproducts). *Replacing one of the* 0 *components of projection* π_2 *(resp.* π_1*) of the standard biproduct (23) by an arbitrary (suitably typed) matrix* C *and the corresponding* 0 *component of* i_1 *(resp.* i_2*) by* $-C$ *yields a biproduct. That is,*

$$\begin{aligned} \pi_1^C &= \left[\begin{array}{c|c} 1 & 0 \end{array}\right] \quad , \quad \pi_2^C = \left[\begin{array}{c|c} C & 1 \end{array}\right] \\ i_1^C &= \left[\frac{1}{-C}\right] \quad , \quad i_2^C = \left[\frac{0}{1}\right] \end{aligned} \tag{47}$$

form a biproduct, parametric on C*, where types are as in the diagram below:*



Proof: *See the appendix.*

Let us inspect the behaviour of the *junc* (16) and *split* (17) combinators arising from this biproduct:

$$\begin{aligned} \left[\begin{array}{c|c} A & B \end{array}\right]^C &= A \cdot \pi_1^C + B \cdot \pi_2^C \\ &= A \cdot \left[\begin{array}{c|c} 1 & 0 \end{array}\right] + B \cdot \left[\begin{array}{c|c} C & 1 \end{array}\right] = \left[\begin{array}{c|c} A & 0 \end{array}\right] + \left[\begin{array}{c|c} B \cdot C & B \end{array}\right] \\ &= \left[\begin{array}{c|c} A + B \cdot C & B \end{array}\right] \\ \left[\frac{A}{B}\right]^C &= i_1^C \cdot A + i_2^C \cdot B \\ &= \left[\frac{1}{-C}\right] \cdot A + \left[\frac{0}{1}\right] \cdot B = \left[\frac{A}{-C \cdot A}\right] + \left[\frac{0}{B}\right] \\ &= \left[\frac{A}{(-C \cdot A) + B}\right] = \left[\frac{A}{B - C \cdot A}\right] \end{aligned} \tag{48}$$

The universal property of *split* will thus be:

$$X = \left[\frac{A}{B}\right]^C \quad \Leftrightarrow \quad \pi_1 \cdot X = A \ \wedge \ \pi_2 \cdot X + C \cdot A = B \tag{49}$$

Note that $\left[\frac{A}{B}\right]^C$ can be recognized as the block-version of an operation common in linear algebra: replacing a row (cf. B) by subtracting from it a multiple of another row (cf. A), as used in Gauss-Jordan elimination. $\left[\begin{array}{c|c} A & B \end{array}\right]^C$ does the same column-wise, adding rather than subtracting.

This enables the following block-version of Gauss-Jordan elimination, where X is supposed to be invertible (always the case if in row-echelon form):

$$\begin{array}{l} gje : (\ k+n \xleftarrow{\quad} k+m\ ) \to (\ k+n \xleftarrow{\quad} k+m\ ) \\ gje \left[\begin{array}{c|c} X & B \\ \hline A & D \end{array}\right] = \left[\begin{array}{c|c} X & B \\ \hline 0 & gje(D - A \cdot X^{-1} \cdot B) \end{array}\right] \\ gje\ X = X \end{array} \tag{50}$$

X^{-1} denotes the inverse of X, that is, $X \cdot X^{-1} = id$ holds. The rationale of the algorithm assumes that the swapping biproduct is first applied as much as needed to transform the input matrix in the form $\left[\begin{array}{c|c} X & B \\ \hline A & D \end{array}\right]$ where topmost-leftmost block X is in row-echelon form. (Listing 3 shows an encoding of (50) into a MATLAB script.) Then the *split* combinator (48) of the self-cancellable biproduct associated to $C = A \cdot X^{-1}$ is used to convert $\left[\begin{array}{c|c} X & B \\ \hline A & D \end{array}\right]$ into a matrix in which cancellation ensures the 0 block of the right-hand side of (50):

$$\begin{array}{rcl} \left[\begin{array}{c} [\ X \mid B\ ] \\ \hline [\ A \mid D\ ] \end{array}\right]^{(A \cdot X^{-1})} & = & \left[\begin{array}{c} [\ X \mid B\ ] \\ \hline [\ A \mid D\ ] - (A \cdot X^{-1}) \cdot [\ X \mid B\ ] \end{array}\right] \\ & = & \left[\begin{array}{c} [\ X \mid B\ ] \\ \hline [\ A \mid D\ ] - [\ A \cdot X^{-1} \cdot X \mid A \cdot X^{-1} \cdot B\ ] \end{array}\right] \\ & = & \left[\begin{array}{c} [\ X \mid B\ ] \\ \hline [\ A - A \mid D - A \cdot X^{-1} \cdot B\ ] \end{array}\right] \\ & = & \left[\begin{array}{c|c} X & B \\ \hline 0 & D - A \cdot X^{-1} \cdot B \end{array}\right] \end{array}$$

The algorithm proceeds recursively applied to (smaller) block $D - A \cdot X^{-1} \cdot B$ until X is found alone, that is, the target type (i.e. number of rows) of A and D is 0.

The classical version of the algorithm corresponds to making block $k \xleftarrow{X} k$ singular, $1 \xleftarrow{x} 1$, yielding

$$\begin{array}{l} ge : (\ 1+n \xleftarrow{\quad} 1+m\ ) \to (\ 1+n \xleftarrow{\quad} 1+m\ ) \\ ge \left[\begin{array}{c|c} x & B \\ \hline A & D \end{array}\right] = \left[\begin{array}{c|c} x & B \\ \hline 0 & ge(D - \frac{A}{x} \cdot B) \end{array}\right] \\ ge\ x = x \end{array} \tag{51}$$

The correction of the algorithm is discussed elsewhere [34] with respect to the specification: *transform the input matrix into one which is in row-echelon (RE) form and keeps the same information*. In brief, (50) ensures RE-form since X is in RE-form (by construction) and $gje(D - A \cdot X^{-1} \cdot B)$ inductively does so. The other requirement is ensured by the universal properties underlying block-notation, granted by the biproduct construction: *splits* and *juncs* are isomorphisms, so they preserve the information of the blocks they put together. For instance, denoting the hom-set of all matrices with n

```
function R = GJE(M)
   [m,n] = size(M);
   k = MPRef(M);
   if k < n
      X = M(1:k,1:k);
      B = M(1:k,k+1:n);
      A = M(k+1:m,1:k);
      D = M(k+1:m,k+1:n);
      R(1:k,1:k) = X;
      R(1:k,k+1:n) = B;
      R(k+1:m,1:k) = zeros(m-(k+1)+1,k);
      R(k+1:m,k+1:n) = GJE(D - A * inv(X) * B);
   else
      R = M;
   end
end
```

Listing 3: MATLAB encoding of the algorithm for blocked version of Gauss-Jordan elimination given by (50). Auxiliary function `MPRef` calculates the size of the largest topmost-leftmost block of input matrix M that is in row-echelon form.

columns and m rows by m^n, property (20) establishes isomorphism

$$m^n \times m^p \quad \cong \quad m^{n+p} \tag{52}$$

— cf. (34) on page 181 of [24]. So, all "juncs" of similarly typed matrices are isomorphic, meaning that they hold the same information under different formats.

Scaling biproducts. Finally, we address the operation of scaling a biproduct by some factor — a device which will be required in the calculational approach to vectorization of Section 8. The question is: given biproduct $m \underset{i_1}{\overset{\pi_1}{\leftrightarrows}} m+n \underset{i_2}{\overset{\pi_2}{\leftrightarrows}} n$, can its dimensions be "scaled up k times"?

This will mean multiplying m and n (and $m+n$) by k. The matrix operation which has this behaviour dimension-wise is the so-called Kronecker product [35]: given $p \xleftarrow{A} m$ and $q \xleftarrow{B} n$, Kronecker product $p \times q \xleftarrow{A \otimes B} m \times n$ is the matrix which replaces each element a_{ij} of A by block $a_{ij}B$.

In the categorial setting of Mat_K, Kronecker product is a tensor product, captured by a (bi)functor $\otimes : Mat_K \times Mat_K \to Mat_K$. On objects, $m \otimes n = m \times n$ (product of two dimensions); on morphisms, $A \otimes B$ is the matrix product defined above. Recall that a category is monoidal [20, 36, 37] when it comes equipped with one such bifunctor which is associative

$$(A \otimes B) \otimes C \quad = \quad A \otimes (B \otimes C) \tag{53}$$

and has a left and a right unit. In the case of Mat_K the unit is id_1. This means that we

can rely on the following properties [10] granting $\otimes$ as a bilinear bifunctor, for suitably typed A, B and C:

$$(A \otimes B) \cdot (C \otimes D) = (A \cdot C) \otimes (B \cdot D) \quad (54)$$

$$id \otimes id = id \quad (55)$$

$$A \otimes (B + C) = (A \otimes B) + (A \otimes C) \quad (56)$$

$$(B + C) \otimes A = (B \otimes A) + (C \otimes A) \quad (57)$$

Theorem 3 (Scaling biproducts). *Let* $m \underset{i_1}{\overset{\pi_1}{\leftrightarrows}} m + n \underset{i_2}{\overset{\pi_2}{\leftrightarrows}} n$ *be a biproduct. Then*

$$m \times k \underset{i_1 \otimes id_k}{\overset{\pi_1 \otimes id_k}{\leftrightarrows}} (m + n) \times k \underset{i_2 \otimes id_k}{\overset{\pi_2 \otimes id_k}{\leftrightarrows}} n \times k$$

is a biproduct.

Proof: *See the appendix.*

This result has a number of nice consequences, namely two simplification rules

$$\pi_j \otimes id = \pi_j \quad (j = 1, 2) \quad (58)$$

$$i_j \otimes id = i_j \quad (j = 1, 2) \quad (59)$$

which lead to the two Kronecker-product fusion laws,

$$\left[\begin{array}{c|c} A & B \end{array}\right] \otimes C = \left[\begin{array}{c|c} A \otimes C & B \otimes C \end{array}\right] \quad (60)$$

$$\left[\frac{A}{B}\right] \otimes C = \left[\frac{A \otimes C}{B \otimes C}\right] \quad (61)$$

which in turn provide for blocked Kronecker product operation. The simplification rules are better understood with types made explicit, for instance

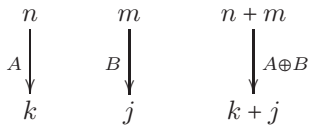
$$(\ n \xleftarrow{\pi_1} m + n\ ) \otimes (\ k \xleftarrow{id} k\ ) = k \times n \xleftarrow{\pi_1} k \times m + k \times n$$

thus exhibiting the type polymorphism of biproduct injections and projections. The calculation of fusion law (60) is given in the appendix and that of (61) is similar.

Finally, we define another Mat_K bifunctor — *direct sum*,

$$A \oplus B = \left[\begin{array}{c|c} i_1 \cdot A & i_2 \cdot B \end{array}\right] \quad (62)$$

of type



[10] More can be said about Mat_K but for our purposes it is enough to stick to its monoidal structure. Further properties can be found in [38]. For alternative definitions of the Kronecker product in terms of other matrix products see section 13.

which is such that

$$id_2 \otimes A \quad = \quad A \oplus A \tag{63}$$

holds. From (62) we see that each biproduct generates its own direct sum. This offers a number of standard properties which can be expressed using coproduct (dually product) combinators. Thus absorption-law

$$\left[\ A \ \middle|\ B\ \right] \cdot (C \oplus D) \quad = \quad \left[\ A \cdot C \ \middle|\ B \cdot D\ \right] \tag{64}$$

and the injections' natural properties which follow:

$$(A \oplus B) \cdot i_1 \quad = \quad i_1 \cdot A \tag{65}$$

$$(A \oplus B) \cdot i_2 \quad = \quad i_2 \cdot B \tag{66}$$

The same properties can be expressed by reversing the arrows, that is, in terms of projections and products. Checking them all from (62) and the universal property of *junc* (dually: *split*) is routine work.

8. Vectorization: "from product to exponentiation"

Vectorization (or linearization) is the operation (linear transformation) which converts a matrix into a (column) vector [11]. Given matrix A below, we can transform it into vector v as shown, which corresponds to parsing A in column-major order :

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \qquad\qquad \mathbf{vec}\ A = \begin{bmatrix} a_{11} \\ a_{21} \\ a_{12} \\ a_{22} \\ a_{13} \\ a_{23} \end{bmatrix}$$

The linearization of an arbitrary matrix into a vector is a data refinement step. This means finding suitable abstraction/representation relations [40] between the two formats and reasoning about them, including the refinement of all matrix operations into vector form. In this section we show that such an abstraction/representation pair is captured by isomorphisms implicit in a universal construct, and use these in calculating the implementation of two matrix combinators — composition and transpose.

8.1. Column-major Vectorization

In the example given above, matrix A is of type $2 \longleftarrow 3$ and $\mathbf{vec}\ A$ is of type $6 \longleftarrow 1$. So, we can write the type of operator **vec** as follows:

$$\mathbf{vec} :: (2 \leftarrow 3 \times 1) \rightarrow (3 \times 2 \leftarrow 1)$$

[11] "Vectorization" is an ambiguous term, for it also means using SIMD vector instructions [39] and not storing matrices as vectors. We adhere to it because of its widespread use in the bibliography, see eg. [3, 33, 35].

Writing 3×1 (resp. 3×2) instead of 3 (resp. 6) is suggestive of the polymorphism of this operator,

$$\mathbf{vec}_k :: (n \leftarrow k \times m) \rightarrow (k \times n \leftarrow m)$$

where a factor k is shunted between the input and the output types.

Thus vectorization is akin to exponentiation, that is, *currying* [17] in functional languages. While currying "thins" the input of a given binary function $f :: c \leftarrow a \times b$ by converting it into its unary (higher-order) counterpart **curry** $f :: (c \leftarrow b) \leftarrow a$, so does vectorization by thinning a given matrix $n \xleftarrow{A} k \times m$ into $k \times n \xleftarrow{\mathbf{vec}\, A} m$.

We will refer to k as the "thinning factor" of the vectorization. This factor is $k = 3$ in the illustration above. For $m = 1$, **vec** A becomes a column vector: the standard situation considered in the literature [33, 35].

As we shall see briefly, operator $\mathbf{vec}_k$ is a bijection, in fact one of the witnesses of the isomorphism that underlies the empirical observation that vectorization and devectorization preserve matrix contents, changing matrix shape only. The other witness is its converse $\mathbf{unvec}_k$:

$$n \leftarrow k \times m \quad \underset{\mathbf{unvec}_k}{\overset{\mathbf{vec}_k}{\cong}} \quad k \times n \leftarrow m$$

As we did for other matrix combinators, we shall capture such intuition formally in the form of the universal property which wraps up the isomorphism above, this time finding inspiration in [41]:

$$X = \mathbf{vec}_k\, A \;\Leftrightarrow\; A = \epsilon_k \cdot (id_k \otimes X) \qquad \begin{array}{c} k \times n \\ {\scriptstyle X}\uparrow \\ m \end{array} \qquad \begin{array}{ccc} k \times (k \times n) & \xrightarrow{\epsilon_k} & n \\ {\scriptstyle id_k \otimes X}\uparrow & \nearrow_{A} & \\ k \times m & & \end{array} \tag{67}$$

Following the standard recipe, (67) grants **vec** and its converse **unvec** as bijective transformations. Among the usual corollaries of (67) we record the following, which will be used shortly: the cancellation-law,

$$A = \epsilon_k \cdot (id_k \otimes \mathbf{vec}_k\, A) \tag{68}$$

obtained for $X := \mathbf{vec}_k\, A$, and a closed formula for devectorization,

$$\mathbf{unvec}\, X \quad = \quad \epsilon \cdot (id \otimes X) \tag{69}$$

obtained from (67) knowing that $X = \mathbf{vec}\, A$ is the same as $\mathbf{unvec}\, X = A$.

For $k = 1$ it is easy to see that vectorization degenerates into identity: $\mathbf{vec}_1\, A = A$ and $\epsilon_1 = id$. We start by putting our index-free, biproduct matrix algebra at work in the calculation of ϵ_k for $k = 2$.

Blocked vectorization. For $k = 2$, the smallest possible case happens for $m = n = 1$, where one expects $\mathbf{vec}_2 \left[\, x \; y \,\right]$ to be $\left[\begin{smallmatrix} x \\ y \end{smallmatrix}\right]$, for x and y elementary data. We proceed to the generalization of this most simple situation by replacing x and y with blocks $n \xleftarrow{A} m$ and $n \xleftarrow{B} m$, respectively, and reasoning:

$$
\begin{array}{ll}
& \mathbf{vec}_2 \left[\begin{array}{c|c} A & B \end{array}\right] = \left[\begin{array}{c} \frac{A}{B} \end{array}\right] \\
\Leftrightarrow & \quad \{ \text{ (67) } \} \\
& \left[\begin{array}{c|c} A & B \end{array}\right] = \epsilon_2 \cdot (id_2 \otimes \left[\begin{array}{c} \frac{A}{B} \end{array}\right]) \\
\Leftrightarrow & \quad \{ \text{ (63) ; unjunc } \epsilon_2 \text{ into } \epsilon_{21} \text{ and } \epsilon_{22} \text{ } \} \\
& \left[\begin{array}{c|c} A & B \end{array}\right] = \left[\begin{array}{c|c} \epsilon_{21} & \epsilon_{22} \end{array}\right] \cdot (\left[\begin{array}{c} \frac{A}{B} \end{array}\right] \oplus \left[\begin{array}{c} \frac{A}{B} \end{array}\right]) \\
\Leftrightarrow & \quad \{ \oplus\text{-absorption (64) } \} \\
& \left[\begin{array}{c|c} A & B \end{array}\right] = \left[\begin{array}{c|c} \epsilon_{21} \cdot \left[\begin{array}{c} \frac{A}{B} \end{array}\right] & \epsilon_{22} \cdot \left[\begin{array}{c} \frac{A}{B} \end{array}\right] \end{array}\right] \\
\Leftrightarrow & \quad \{ \text{ (33) ; (22) } \} \\
& \epsilon_{21} = \pi_1 \wedge \epsilon_{22} = \pi_2 \\
\Leftrightarrow & \quad \{ \text{ junc } \epsilon_{21} \text{ and } \epsilon_{22} \text{ back into } \epsilon_2 \text{ } \} \\
& \epsilon_2 = \left[\begin{array}{c|c} \pi_1 & \pi_2 \end{array}\right]
\end{array}
$$

We have obtained, with types

$$n \xleftarrow{\epsilon_2} 2n + 2n \;=\; \left[\begin{array}{c|c} \pi_1 & \pi_2 \end{array}\right] \tag{70}$$

expressing ϵ_k (for $k = 2$) in terms of the standard biproduct projections. Thus $\mathbf{vec}_2 \, \epsilon = \left[\begin{smallmatrix} \pi_1 \\ \pi_2 \end{smallmatrix}\right] = id_2$, a particular case of *reflection law*,

$$\mathbf{vec}_k \, \epsilon_k \;=\; id_{k \times n} \tag{71}$$

easy to obtain in general from (67) by letting $X := id_{k \times n}$ and simplifying. This can be rephrased into

$$\epsilon \;=\; \mathbf{unvec}\, id \tag{72}$$

providing a generic way of defining the mediating matrix ϵ in (67).

As an exercise, we suggest the reader checks the following instance of cancellation law (72), for $k = m = n = 2$:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot (id_2 \otimes \begin{bmatrix} a_{11} \\ a_{21} \\ a_{12} \\ a_{22} \end{bmatrix})$$

It can be observed that $\epsilon = \left[\begin{smallmatrix} 1&0&0&0&0&0&1&0 \\ 0&1&0&0&0&0&0&1 \end{smallmatrix}\right] = \begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix} \otimes id_2$. This illustrates equality

$$\epsilon \otimes id \quad = \quad \epsilon \tag{73}$$

easy to draw from previous results [12]. Again rendering types explicit helps in checking what is going on:

$$(\ k^2 \times n \xrightarrow{\epsilon_k} n\ ) \otimes (\ j \xrightarrow{id} j\ ) \quad = \quad k^2 \times (n \times j) \xrightarrow{\epsilon_k} n \times j$$

Doing a similar exercise for $k = 3$, $\mathbf{vec}_3 \begin{bmatrix} A & B & C \end{bmatrix} = \begin{bmatrix} A \\ B \\ C \end{bmatrix}$ — that is,

$$\mathbf{vec}_3 \left[\ \left[\ A \ \middle|\ B\ \right] \middle|\ C\ \right] = \left[\begin{array}{c} \left[\begin{array}{c} A \\ \hline B \end{array}\right] \\ \hline C \end{array}\right]$$

— one would obtain for $3^2 \times n \xrightarrow{\epsilon_3} n$ matrix $\left[\ \left[\ \pi_1 \cdot \pi_1 \ \middle|\ \pi_2 \cdot \pi_1\ \right] \middle|\ \pi_2\ \right]$, with types as in diagram:

$$\begin{array}{ccccccc} n & \xleftarrow{\pi_1} & n+n & \xleftarrow{\pi_1} & (n+n)+n & \xrightarrow{\pi_2} & n \\ & & \downarrow{\scriptstyle \pi_2} & & & & \\ & & n & & & & \end{array}$$

Recalling absorption law (64), (63) and ϵ_2 (70), we observe that ϵ_3 rewrites to $\left[\ \left[\ \pi_1 \ \middle|\ \pi_2\ \right] \cdot (\pi_1 \oplus \pi_1) \ \middle|\ \pi_2\ \right]$, itself the same as $\left[\ \epsilon_2 \cdot (id_2 \otimes \pi_1) \ \middle|\ \pi_2\ \right]$, providing a hint of the general case:

$$\epsilon_{k+1} \quad = \quad \left[\ \epsilon_k \cdot (id_k \otimes \pi_1) \ \middle|\ \pi_2\ \right] \tag{74}$$

$$\epsilon_1 \quad = \quad id \tag{75}$$

Let us typecheck (74), assuming completely independent types as starting point:

$$j \xleftarrow{\epsilon_{k+1}} (k+1)^2 \times j$$

$$i \xleftarrow{\epsilon_k} k^2 \times i$$

$$k \xleftarrow{id_k} k$$

$$n \xleftarrow{\pi_1} n + m$$

$$b \xleftarrow{\pi_2} a + b$$

[12] See the appendix. Equality (73) provides an explanation for the index-wise construction of ϵ given in [41].

Type equations $a = n$ and $b = m$ follow from π_1 and π_2 belonging to the same biproduct. Term $\epsilon_k \cdot (id_k \otimes \pi_1)$ forces type equation ("unification") $k^2 \times i = k \times n$, that is, $n = k \times i$. Term $[\ \epsilon_k \cdot (id_k \otimes \pi_1)\ |\ \pi_2\ ]$ entails $i = m$. Finally, the whole equality forces

$$\begin{aligned} j &= m \\ (k+1)^2 \times j &= k \times (k \times i + m) + m + k \times i \end{aligned}$$

whereby, unfolding and substituting, $k^2 \times m + 2k \times m + m = k^2 \times i + k \times m + i + k \times i$ yields $i = m$. Thus the most general types of the components of (74) are:

$$m \xleftarrow{\epsilon_{k+1}} (k+1)^2 \times m$$

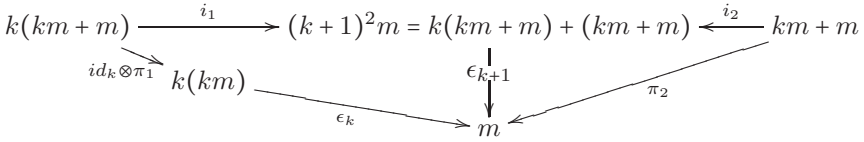
$$m \xleftarrow{\epsilon_k} k^2 \times m$$

$$k \xleftarrow{id_k} k$$

$$k \times m \xleftarrow{\pi_1} k \times m + m$$

$$m \xleftarrow{\pi_2} k \times m + m$$

as displayed in the following diagram (dropping × symbols for better layout):



From (69) and looking at the diagram above we find an even simpler way of writing (74):

$$\epsilon_{k+1} = [\ \mathbf{unvec}_k\, \pi_1\ |\ \pi_2\ ]$$

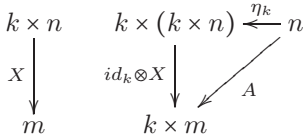
Summing up, the index-free definition of the counit ϵ_k of a vectorization for any thinning factor k is made possible by use of the biproduct construction, by induction on k. The corresponding encoding in MATLAB can be found in Listing 4.

8.2. Devectorization

There is another way of characterizing column-major vectorization, and this is by reversing the arrows of (67) and expressing the universal property of **unvec**, rather than that of **vec**,

$$X = \mathbf{unvec}_k\, A \ \Leftrightarrow\ A = (id_k \otimes X) \cdot \eta_k \tag{76}$$

cf. diagram



where (dropping subscripts) $\eta = \epsilon^\circ$ [41]. From this we infer not only the cancellation-law of devectorization,

$$A = (id \otimes \mathbf{unvec}\, A) \cdot \eta \tag{77}$$

```
function E = epsilon(k,m)
   if k==1
     E = eye(m)
   else
     n=k-1;
     p1 = eye(n*m,k*m);
     p2 = jay(m,k*m);
     E = [ (epsilon(n,m) * kron(eye(n),p1)) p2 ]
   end
end

function J = jay(r,c)
       if r>=c
         J = [ zeros(r-c,c) ; eye(c) ];
       else
         J = [ zeros(r,c-r) eye(r) ];
       end
end
```

Listing 4: MATLAB encoding of ϵ_k (74,75). Auxiliary function `jay` (cf. letter J) implements biproduct components π_2 and i_2 in the same way `eye` (cf. letter I) implements π_1 and i_1. Both `eye` and `kron` are primitive operations in MATLAB providing the identity matrix and the Kronecker product operation.

but also a closed formula for vectorization,

$$\mathbf{vec}\, X \quad = \quad (id \otimes X) \cdot \eta \tag{78}$$

since $X = \mathbf{unvec}\, A$ is the same as $\mathbf{vec}\, X = A$. Thus

$$\eta_k \quad = \quad \mathbf{vec}_k\, id_{k\times m} \tag{79}$$

holds. Reversing the arrows also entails the following converse-duality,

$$(\mathbf{vec}\, A)^\circ \quad = \quad \mathbf{unvec}\, (A^\circ) \tag{80}$$

easy to draw from (69) and (78).

8.3. Self-adjunction

Summing up, we are in presence of an adjunction between functor $\mathsf{F}X = id_k \otimes X$ and itself — a *self-adjunction* [41] — inducing a monoidal closed structure in the category. The root for this is again the biproduct, entailing the same functor $\oplus$ (62) for both coproduct and product. It is known that the latter is the right adjoint of the diagonal functor $\Delta(n) = (n, n)$, which in turn is the right adjoint of the former. Using adjunction's notation, $\oplus \dashv \Delta$ and $\Delta \dashv \oplus$ hold. By adjunction composition [20] one obtains $(\oplus{\cdot}\Delta) \dashv (\oplus{\cdot}\Delta)$, whereby — because $\oplus{\cdot}\Delta = (id_2\otimes)$ (63) — the self-adjunction $(id_2\otimes) \dashv (id_2\otimes)$ holds.

9. Unfolding vectorization algebra

This section will show how vectorization theory, as given in eg. [33, Chap. 10], follows from universal properties (67,77) by index-free calculation. This is an advance over the traditional, index-wise matrix representations and proofs [33, 35, 41] where notation is often quite loose, full of dot-dot-dots. We will also stress on the role of matrix types in the reasoning.

Vectorization is linear. To warm up let us see a rather direct result, the linearity of **vec** :

$$\mathbf{vec}\,(A+B) \quad = \quad \mathbf{vec}\,A + \mathbf{vec}\,B \tag{81}$$

Its derivation, which is a standard exercise in algebra-of-programming calculation style, can be found in the appendix.

Roth's relationship. On the other side of the spectrum we find the following relationship of the **vec** operator and Kronecker product

$$\mathbf{vec}\,(A \cdot B \cdot C) \quad = \quad (C^{\circ} \otimes A) \cdot \mathbf{vec}\,B \tag{82}$$

which Abadir and Magnus [33] attribute to Roth [42] and regard as the fundamental result of the whole theory.

In [33], (82) is said to hold "whenever the product ABC is defined". Our typed approach goes further in enabling us to find the *most general* type which accommodates the equality. The exercise is worthwhile detailing in so far as it spells out two different instances of polymorphic **vec** , with different thinning-factors. We speed up the inference by starting from types already equated by the matrix compositions and by the equality as a whole:

$$\begin{array}{l} j \xleftarrow{\;A\;} n \xleftarrow{\;B\;} k \xleftarrow{\;C\;} m \\ m \times j \xleftarrow{\;C^{\circ} \otimes A\;} k \times n \xleftarrow{\;\mathbf{vec}\,B\;} x \\ m \times j \xleftarrow[\;\mathbf{vec}\,(A \cdot B \cdot C)\;]{} x \end{array}$$

The type relationship between B and $\mathbf{vec}\,B$ entails $k = k \times x$, and therefore $x = 1$. Thus the *principal type* of (82) is:

$$\begin{array}{l} j \xleftarrow{\;A\;} n \xleftarrow{\;B\;} k \xleftarrow{\;C\;} m \\ m \times j \xleftarrow{\;C^{\circ} \otimes A\;} k \times n \xleftarrow{\;\mathbf{vec}_k\,B\;} 1 \\ \qquad\qquad\qquad\qquad \xleftarrow[\;\mathbf{vec}_m\,(A \cdot B \cdot C)\;]{} \end{array}$$

We will show briefly that (82) is the merge of two other facts which express the vectorization of the product of two matrices B and C in two alternative ways,

$$\mathbf{vec}_k\,(B \cdot C) \quad = \quad (id_k \otimes B) \cdot \mathbf{vec}_k\,C \tag{83}$$

$$\mathbf{vec}_m\,(C \cdot B) \quad = \quad (B^{\circ} \otimes id_n) \cdot \mathbf{vec}_k\,C \tag{84}$$

whose types schemes are given by diagrams

$$\begin{array}{ccccc} j & \xleftarrow{B} & n & \xleftarrow{C} & k \times m \\ k \times j & \xleftarrow{id_k \otimes B} & k \times n & \xleftarrow{\mathbf{vec}_k\, C} & m \end{array} \qquad \mathbf{vec}_k\,(B \cdot C) \tag{85}$$

and

$$\begin{array}{ccccc} & & n & \xleftarrow{C} & k & \xleftarrow{B} & m \\ m \times n & \xleftarrow{B^\circ \otimes id_n} & k \times n & \xleftarrow{\mathbf{vec}_k\, C} & 1 & & \end{array} \qquad \mathbf{vec}_m\,(C \cdot B)$$

respectively. The derivation of (83) follows by instantiation of cancellation law (77), for $A := \mathbf{vec}\,(B \cdot C)$, knowing that $\mathbf{unvec}\,(\mathbf{vec}\,X) = X$ — see the appendix. The calculation of (84), also in the appendix, proceeds in the same manner. Thanks to these two results, calculating (82) is routine work:

$$\begin{array}{ll} & (C^\circ \otimes A) \cdot \mathbf{vec}\,B \\ = & \quad \{ \text{ identity ; bifunctor } \otimes \} \\ & (C^\circ \otimes id) \cdot (id \otimes A) \cdot \mathbf{vec}\,B \\ = & \quad \{ \text{ (83) } \} \\ & (C^\circ \otimes id) \cdot \mathbf{vec}\,(A \cdot B) \\ = & \quad \{ \text{ (84) } \} \\ & \mathbf{vec}\,(A \cdot B \cdot C) \end{array}$$

Roth's relationship (82) is proved in different ways in the literature. In [35], for instance, it turns up in the proof of a result about the *commutation matrix* which will be addressed in the following section. In [33] it is calculated by expressing matrix B as a summation of vector compositions and relying on the linearity of **vec**, using an auxiliary result about Kronecker product of vectors. In a similar approach, a proof for the particular case of boolean matrices is presented in [3] using relational product.

Vectorization as (blocked) transposition. Finally, we state a result which relates vectorization with transposition — compare with (25):

$$\mathbf{vec}_{k+k'} \left[\!\!\left[\begin{array}{c|c} A & B \end{array} \right]\!\!\right] = \left[\!\!\left[\begin{array}{c} \mathbf{vec}_k\, A \\ \hline \mathbf{vec}_{k'}\, B \end{array} \right]\!\!\right] \tag{86}$$

Type inference reveals that the most generic types which accommodate this result are $n \xleftarrow{A} k \times m$ and $n \xleftarrow{B} k' \times m$. The proof can be found in the appendix.

When does, then, vectorization (86) *coincide* with transposition (25)? We reason:

$$\begin{array}{ll} & \mathbf{vec}_{k+k'} \left[\begin{array}{c|c} A & B \end{array}\right] = \left[\begin{array}{c|c} A & B \end{array}\right]^\circ \\ \Leftrightarrow & \quad \{\ (25)\ ;\ (86)\ ;\ (34)\ \} \\ & \mathbf{vec}_k\ A = A^\circ\ \wedge\ \mathbf{vec}_{k'}\ B = B^\circ \end{array}$$

The two clauses correspond to the induction hypothesis in a structurally inductive argument, breaking down thinning factors until base case $k = 1$ is reached. Since $\mathbf{vec}_1\ X = X$, we conclude that vectorization *is* transposition wherever A and B can be broken in "rows" of symmetric blocks, that is, blocks X such that $X = X^\circ$. In the particular case of $\left[\begin{array}{c|c} A & B \end{array}\right]$ being a row vector (type $n = 1$), this always happens, the symmetric blocks being individual cells of type $1 \xleftarrow{\quad} 1$. Thus

$$\mathbf{vec}_m\ A \quad = \quad A^\circ \tag{87}$$

holds, for $1 \xleftarrow{A} m$ a row vector.

10. Calculating vectorized operations

We close the paper by showing how typed linear algebra helps in calculating matrix operations in vectorial form. We only address the two basic combinators *transpose* and *composition*, leaving aside the sophistication required by the parallel implementation of such combinators. (See eg. [13, 14, 15, 16, 39] concerning the amazing evolution of the subject in recent years.)

To begin with, let us show that transposition can be expressed solely in terms of the **vec** and **unvec** combinators. The argument is a typical example of reasoning with arrows in a categorial framework. Let $n \xleftarrow{A} m$ be an arbitrary matrix. We start by building $1 \xleftarrow{B=\mathbf{unvec}_n\ A} n \times m$, then $n \times m \xleftarrow{C=\mathbf{vec}_{n\times m}\ B} 1$ and, finally $m \xleftarrow{\mathbf{unvec}_n\ C} n$. So, the outcome $\mathbf{unvec}_n\ (\mathbf{vec}_{n\times m}\ (\mathbf{unvec}_n\ A))$ has the same type as A°. Checking that they are actually the same arrow is easy, once put in another way:

$$\mathbf{vec}_n\ (A^\circ) \quad = \quad \mathbf{vec}_{n\times m}\ (\mathbf{unvec}_n\ A) \tag{88}$$

We calculate:

$$\begin{array}{ll} & \mathbf{vec}_n\ (A^\circ) \\ = & \quad \{\ (80)\ \} \\ & (\mathbf{unvec}_n\ A)^\circ \\ = & \quad \{\ (87) \text{ since } \mathbf{unvec}_n\ A \text{ is of type } 1 \xleftarrow{\quad} m \times n\ \} \\ & \mathbf{vec}_{m\times n}\ (\mathbf{unvec}_n\ A) \end{array}$$

Next, we show how (88) helps in calculating a particular, generic matrix — the *commutation matrix* — usefull to implement transposition of matrices encoded as vectors using matrix-vector products.

10.1. Implementing transposition in vectorial form

Magnus and Neudecker [35] present the *commutation matrix* $n \times m \xleftarrow{K_{nm}} m \times n$ as the unique solution K_{nm} to equation

$$\mathbf{vec}_n\,(A^\circ) = K_{nm} \cdot \mathbf{vec}_m\,A \qquad \text{cf. diagram} \qquad \begin{array}{ccccc} n \times m & \xleftarrow{K_{nm}} & m \times n & & n \\ & \nwarrow^{\mathbf{vec}_n\,A^\circ} & \uparrow{\scriptstyle \mathbf{vec}_m\,A} & & \uparrow{\scriptstyle A} \\ & & 1 & & m \end{array} \tag{89}$$

the practical impact of which is obvious: knowing how to build (generic) K_{nm} enables one to transpose matrix A by composing K_{nm} with A vectorized, the outcome being delivered as a vector too. Implemented in this way, transposition can take advantage of the *divide-and-conquer* nature of matrix multiplication and therefore be efficiently performed on parallel machines.

The uniqueness of K_{nm} is captured by the "universal" property,

$$X = K_{nm} \quad \Leftrightarrow \quad \mathbf{vec}\,(A^\circ) = X \cdot \mathbf{vec}\,A \tag{90}$$

of which (89) is the cancellation corollary. However, (90) defines K_{nm} implicitly, not its explicit form. In the literature, this matrix (also referred to as the *stride permutation* matrix [13, 29]) is usually given using indexed notation. For instance, Magnus and Neudecker [35] define it as a double summation

$$K_{nm} \quad = \quad \sum_{i=1}^{n} \sum_{j=1}^{m} (H_{ij} \otimes H_{ij}^\circ) \tag{91}$$

where each component H_{ij} is a (n, m) matrix with a 1 in its ijth position and zeros elsewhere.

Below we give a simple calculation of its generic formula, arising from putting (89) and (88) together:

$$K_{nm} \cdot \mathbf{vec}_m\,A \quad = \quad \mathbf{vec}_{n\times m}\,(\mathbf{unvec}_n\,A)$$

Knowing the reflection law $\mathbf{vec}_m\,\epsilon_m = id$ (71) and substituting we obtain a closed formula for the commutation matrix:

$$K_{nm} \quad = \quad \mathbf{vec}_{n\times m}\,(\mathbf{unvec}_n\,\epsilon_m) \tag{92}$$

The types involved in this formula can be traced as follows: take $id_{m\times n}$ and devectorize it, obtaining $n \xleftarrow{\epsilon_m} m \times (m \times n)$. Then devectorize this again, getting $1 \xleftarrow{\mathbf{unvec}_n\,\epsilon_m} (n \times m) \times (m \times n)$. Finally, vectorize this with the product of the two thinning factors m and n, to obtain $n \times m \xleftarrow{K_{nm}=\mathbf{vec}_{n\times m}\,(\mathbf{unvec}_n\,\epsilon_m)} m \times n$.

```
function R = cm(n,m)
   R = Vec(n*m,UnVec(n,epsilon(m,n)));
end

function R = cmx(n,m)
  a=kron(eye(n*m),epsilon(n,1));
  b=kron(eye((n*m)*n),epsilon(m,n));
  R = a*b*eta(n*m,m*n);
end
```

Listing 5: Two MATLAB encodings of commutation matrix K_{mn} following (92) and its expansion (93).

The conceptual economy of (92) when compared with (91) is beyond discussion. A factorization of (92) can be obtained by unfolding the **vec** and **unvec** isomorphisms:

$$\begin{aligned} K_{nm} &= \mathbf{vec}_{n\times m}\,(\mathbf{unvec}_n\,\epsilon_m) \\ &= (id_{n\times m}\otimes(\epsilon_n\cdot(id_n\otimes\epsilon_m)))\cdot\eta_{n\times m} \\ &= (id_{n\times m}\otimes\epsilon_n)\cdot(id_{n\times m}\otimes(id_n\otimes\epsilon_m))\cdot\eta_{n\times m} \\ &= (id_{n\times m}\otimes\epsilon_n)\cdot(id_{(n\times m)\times n}\otimes\epsilon_m)\cdot\eta_{n\times m} \end{aligned} \tag{93}$$

Listing 5 includes both versions of the commutation matrix encoded in MATLAB notation.

Magnus and Neudecker [35] give many properties of the commutation matrix, including for instance,

$$(B\otimes A)\cdot K_{ts} = K_{nm}\cdot(A\otimes B) \tag{94}$$

which in our categorial setting is nothing but the statement of its *naturality* in the underlying category of matrices (polymorphism), cf. diagram:

$$\begin{array}{ccc} t\times s & \xleftarrow{K_{ts}} & s\times t \\ {\scriptstyle B\otimes A}\downarrow & & \downarrow{\scriptstyle A\otimes B} \\ n\times m & \xleftarrow[K_{nm}]{} & m\times n \end{array}$$

Another property, not given in [35],

$$K_{nn}\cdot\eta_n = \eta_n \tag{95}$$

is easy to draw from (89) — just let $A := id_n$ and simplify.

10.2. Implementing MMM under matrix-to-vector representation

As we did for transpose, let us reuse previous results in refining MMM to vectorized form. Applying (68) to B in (83) we obtain, recalling type scheme (85):

$$\mathbf{vec}_k\,(B\cdot C) = (id_k\otimes(\epsilon_n\cdot(id_n\otimes\mathbf{vec}_n\,B)))\cdot\mathbf{vec}_k\,C$$

```
function vBC = vecMMM(n,vB,vC)

   a = length(vB);
   b = length(vC);

   if(mod(a,n) ~= 0 || mod(b,n) ~= 0)
      error('n must be a common length factor');
   else
      j = a / n;
      k = b / n;
      x=kron(eye(k),epsilon(n,j));
      y=kron(eye(k*n),vB);
      vBC = ap(x,ap(y,vC));
   end
end
```

Listing 6: MATLAB encoding of vectorized MMM. Intermediate type n is given explicitly. `vB` and `vC` are input vectors which represent composable matrices B and C, respectively. Matrix x is a constant matrix which can be made available at compile time. $ap(B, v)$ denotes the application of matrix B to input vector v.

This re-writes to

$$\mathbf{vec}_k\,(B \cdot C) \quad = \quad (id_k \otimes \epsilon_n) \cdot (id_{k\times n} \otimes \mathbf{vec}_n\, B) \cdot \mathbf{vec}_k\, C \tag{96}$$

It may seem circular to resort to composition in the right hand side of the above, but in fact all instances of composition there are of a special kind: they are matrix-vector products, cf. linear signal transforms [29]. Denoting such an application of a matrix B to a vector v by $ap(B, v)$, we can encode (96) into MATLAB function `vecMMM` shown in Listing 6, under type scheme:

$$\begin{array}{ccccccc} j & \xleftarrow{\;B\;} & n & \xleftarrow{\;C\;} & k & & \\ k\times j & \xleftarrow{\;id_k\otimes\epsilon_n\;} & k\times(n\times j) & \xleftarrow{\;id_{k\times n}\otimes vB\;} & k\times n & \xleftarrow{\;vC\;} & 1 \\ & & & \underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{vBC} & & & \end{array}$$

The operator equivalent to this in the Operator Language DSL of [16] has interface

$$MMM_{j,n,k} \quad : \quad \mathbb{R}^{jn} \times \mathbb{R}^{nk} \to \mathbb{R}^{jk}$$

assuming the field of real numbers and vectors representing matrices in row-major order. The operator is specified by a number of breakdown rules expressing recursive divide-and-conquer algorithmic strategies.

For instance, one such rule prescribes the divide-and-conquer algorithm that splits the left-hand vector row-wise in a number of blocks. Instantiating the rule to the particular case of a two-block split corresponds to our law (26), vectorized. The whole

OL syntax is very rich and explaining its intricacies in the current paper would be a long detour. See Section 13 for on-going work on typing OL formulæ according to the principles advocated in the current paper.

11. Related Work

Categories of matrices can be traced back to the works of MacLane and Birkhoff [20, 24], with focus on either illustrating additive categories or establishing a relationship between linear transformations and matrices. Biproducts have been extensively studied in algebra and category theory. In [24], the same authors find applications of biproducts to the study of additive Abelian groups and modules. A relationship between biproducts and matrices can also be found in [24], but it is nevertheless in [20] that the hint which triggered the current paper can be found explicit (recall Section 4). However, no effort on exploiting biproducts calculationally is present, let alone algorithm derivation. To the best of the authors' knowledge, the current paper presents the first effort to put biproducts in the place they deserve in matrix algebra.

Bloom et al [5] define a generic notion of *machine* and give their semantics in terms of categories of matrices, under special (blocked) composition schemes. They make implicit use of what we have identified as the standard biproduct (enabling blocked matrix algebra) to formalize column and row-wise matrix join and fusion, but the emphasis is on iteration theories which matricial theories are a particular case of.

Other categorial approaches to linear algebra include relative monads [43], whereby the category of finite-dimensional vector spaces arises as a kind of Kleisli category. Efforts by the mathematics of program construction community in the derivation of matrix algorithms include the study of two-dimensional pattern matching [44].

Reference [5] is related to Kleene algebras of matrices [45]. An account of the work on calculational, index-free reasoning about regular and Kleene algebras of matrices can be found in [1]. The close relationship between categories of matrices and relations is implicit in the allegorial setting of Freyd and Ščedrov [19]: essentially, matrices whose data values are taken from *locales* (eg. the Boolean algebra of truth values) are the morphisms of the corresponding allegory (eg. that of binary relations). Bird and de Moor [18] follow [19]. Schmidt [3] dwells on the same relation-matrix binomial relationship too, but from a different, set-theoretical angle. He nevertheless pushes it quite far, eg. by developing a theory of vectorization in relation algebra. Relational biproducts play no explicit role in either [3], [19] or [18].

12. Conclusions

In this paper we have exploited the formalization of matrices as categorial morphisms (arrows) in a way which relates categories of matrices to relation algebra and program calculation. Matrix multiplication is dealt with in detail, in an elegant, calculational style whereby its divide-and-conquer, triple-nested-loop and vectorized implementations are derived.

The notion of a categorial biproduct is at the heart of the whole approach. Using categories of matrices and their biproducts we have developed the algebra of matrix-block operations and shown how biproducts scale up so as to be fit for particular applications of linear algebra such as Gaussian elimination, for instance.

We have also shown how matrix-categorial biproducts shed light into the essence of an important data transformation — vectorization — indispensable to the efficient implementation of linear algebra packages in parallel machines. Our calculations in this respect have shown how polymorphic standard matrices such as eg. the *commutation matrix* are, making dimension polymorphism an essential part of the game, far beyond the loose *"valid only for matrices of the same order"* [33] attitude found in the literature. We have prototyped our constructs and diagrams in MATLAB™ all the way through, and this indeed showed how tedious and error-prone it is to keep track of matrix dimensions in complex expressions. It would be much nicer to write eg. `eye` instead of `eye(n)`, for some hand-computed n and let MATLAB infer which n accommodate the formula we are writing.

The prospect of building biproduct-based type checkers for computer algebra systems such as MATLAB is therefore within reach. This seems to be already the approach in Cryptol [46], a Haskell based DSL for cryptography, where array dimensions are inferred using a strong type-system based on Hindley-Milner style polymorphism extended with arithmetic size constraints.

In retrospect, we believe to have contributed to a better understanding of the blocked nature of linear algebra notation, which is perhaps its main advantage — the *tremendous improvement in concision* which Backhouse stresses in the quotation which opens the paper — and which can be further extended thanks to the (still to be exploited) algebra of biproducts. This raises the issue of matrix polymorphism and enriches our understanding that matrix dimensions are more than just numbers: they are types in the whole sense of the word. Thus the matrix concept spruces up, raising from the untyped number-container view ("rectangles of numbers") to the typed hom-set view in a category. Perhaps Sir Arthur Eddington (1882-1944) was missing this richer view when he wrote [13]:

> *I cannot believe that anything so ugly as multiplication of matrices is an essential part of the scheme of nature* [47, page 39].

13. Future Work

A comprehensive calculational approach to linear algebra algorithm specification, transformation and generation is still missing. However, the successes reported by the engineering field in automatic library generation are a good cue to the feasibility of such a research plan. We intend to contribute to this field of research in several directions.

[13] The authors are indebted to Jeremy Gibbons for pointing their attention to this interesting remark of the great physicist.

SPIRAL. The background of our project is the formalization of OL, the *Operator Language* of [14, 16], in matrix-categorial biproduct terms. In the current paper we have stepped forward in this direction (as compared to [21], for instance) in developing a categorial approach to vectorization, but much more work is still needed to achieve a complete account of the refinement steps implicit in all OL-operator breakdown rules. SPIRAL's row-major vectorized representation calls for further work in adapting our results to such a variant of vectorization.

Kleene algebras of matrices. Thus far we have assumed matrices to take their elements from an algebraic field K. The matrix concept, however, extends to other, less rich algebraic structures, typically involving semirings instead of rings, for instance. Fascinating work in this wider setting shows how, by Kleene algebra, some graph algorithms are unified into Gaussian elimination, for instance [1]. We would thus like to study the impact of such a relaxation on our biproduct approach to the same algorithm.

Khatri-Rao product generalization of relational forks. The monoidal structure provided by the tensor product defined in Section 7 is the key concept to generalize, to arbitrary matrices, the relational (direct) product presented in [3, 26].

It turns out that the *fork* operation of relation algebras [26] is nothing but the operator known in the linear algebra community as the Khatri-Rao product [48]. The standard definition offers this product as a (column-wise) variant of Kronecker product. To emphasise the connection to relation algebra, our definition is closer to that of a fork [26]: given matrices $m \xleftarrow{A} n$ and $k \xleftarrow{B} n$, the Khatri-Rao product (fork) of A and B, denoted $A \triangledown B$, is the matrix of type $m \times k \longleftarrow n$ defined by

$$A \triangledown B = (p_1{}^\circ \cdot A) * (p_2{}^\circ \cdot B)$$

where $A * B$ is the Hadamard (element-wise) product and matrices $m \xleftarrow{p_1} m \times k$ and $k \xleftarrow{p_2} m \times k$ are known as *projections*. To define these we rely on row vectors wholly filled up with 1s, denoted by symbol "!" [14]:

$$\begin{aligned} p_1 &= id \otimes ! \\ p_2 &= ! \otimes id \end{aligned}$$

Khatri-Rao product is associative and its unit is !, that is, $! \triangledown A = A = A \triangledown !$ hold. The close link between the Khatri-Rao and Kronecker products can be appreciated by expressing the latter in terms of the former, $A \otimes B = (A \cdot p_1) \triangledown (B \cdot p_2)$, that is,

$$A \otimes B = (p_1{}^\circ \cdot A \cdot p_1) * (p_2{}^\circ \cdot B \cdot p_2)$$

meaning that Khatri-Rao can be used as alternative to Kronecker in formulating concepts such as, for instance, vectorization [3]. A thorough comparison of both approaches in the setting of arbitrary matrices is a topic for future work [34].

[14] Notation "!" is imported from the algebra of programing [18].

Self-adjunctions. The self-adjunction which supports our approach to vectorization offers a *monad* which we have not yet exploited. Already in (76) we see the unit $k^2 \times n \xleftarrow{\eta} n$ at work, for functor $\mathsf{T}\ n = k^2 \times n$, whose multiplication is of type $k^2 \times n \xleftarrow{\mu} k^4 \times n$ and can be computed following the standard theory:

$$\mu \quad = \quad id \otimes \mathbf{unvec}\, id \ = \ id \otimes \epsilon$$

Curiously enough, the monadic flavour of vectorization can already be savored in version (96) of MMM, suggesting such an implementation as analogue to composition in the "brother" Kleisli category:

$$B \bullet A \quad = \quad \underbrace{(id_k \otimes \epsilon_n)}_{\mu} \cdot \underbrace{(id_{k \times n} \otimes B)}_{\mathsf{F}\ B} \cdot A$$

This should be studied in detail, in particular concerning the extent to which known laws of vectorization are covered by the generic theory of monads, discharging the corresponding proof obligations. The relationship between this monadic setting and that of *relative monads* presented in [43] is another stimulus for further work in this research thread.

Acknowledgements.

The authors would like to thank Markus Püschel (CMU) for driving their attention to the relationship between linear algebra and program transformation. Hugo Macedo further thanks the SPIRAL group for granting him an internship at CMU.

Thanks are also due to Michael Johnson and Robert Rosebrugh (Macquarie Univ.) for pointing the authors to the categories of matrices approach. Yoshiki Kinoshita (AIST, Japan) and Manuela Sobral (Coimbra Univ.) helped with further indications in the field.

This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-010047. Hugo Macedo holds FCT grant number SFRH/BD/33235/2007.

References

[1] R. Backhouse, Mathematics of Program Construction, University of Nottingham, 2004, draft of book in preparation. 608 pages.

[2] D. L. Parnas, Really rethinking "formal methods", IEEE Computer 43 (1) (2010) 28–34.

[3] G. Schmidt, Relational Mathematics, no. 132 in Encyclopedia of Mathematics and its Applications, Cambridge University Press, 2010.

[4] R. Maddux, The origin of relation algebras in the development and axiomatization of the calculus of relations, Studia Logica 50 (3/4) (1991) 421–455.

[5] S. L. Bloom, N. Sabadini, R. F. C. Walters, Matrices, machines and behaviors, Applied Categorical Structures 4 (4) (1996) 343–360.

[6] J. Conway, Regular Algebra and Finite Machines, Chap. & Hall, London, 1971.

[7] N. Trčka, Strong, weak and branching bisimulation for transition systems and Markov reward chains: A unifying matrix approach, in: S. Andova, *et al* (Eds.), Proceedings First Workshop on Quantitative Formal Methods: Theory and Applications, Vol. 13 of EPTCS, 2009, pp. 55–65.

[8] A. Silva, F. Bonchi, M. M. Bonsangue, J. J. M. M. Rutten, Quantitative Kleene coalgebras, Inf. Comput. 209 (5) (2011) 822–849.

[9] A. Sernadas, J. Ramos, P. Mateus, Linear algebra techniques for deciding the correctness of probabilistic programs with bounded resources, Tech. rep., SQIG-IT and TU Lisbon, 1049-001 Lisboa, Portugal (2008).

[10] M. Baroni, R. Zamparelli, Nouns are vectors, adjectives are matrices: representing adjective-noun constructions in semantic space, in: Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing, EMNLP '10, Association for Computational Linguistics, Morristown, NJ, USA, 2010, pp. 1183–1193.

[11] B. Coecke, M. Sadrzadeh, S. Clark, Mathematical foundations for a compositional distributed model of meaning, Linguistic Analysis 36 (1-4) (2010) 345–384.

[12] H. Macedo, J. Oliveira, Do the middle letters of "OLAP" stand for linear algebra ("LA")?, journal paper (submitted) (2011).

[13] J. R. Johnson, R. W. Johnson, D. Rodriguez, R. Tolimieri, A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures, Circuits Syst. Signal Process. 9 (4) (1990) 449–500.

[14] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, N. Rizzolo, SPIRAL: Code generation for DSP transforms, Proceedings of the IEEE 93 (2) (2005) 232–275.

[15] R. A. V. de Geijn, E. S. Quintana-Ortí, The Science of Programming Matrix Computations, `www.lulu.com`, 2008.

[16] F. Franchetti, F. de Mesmay, D. McFarlin, M. Püschel, Operator language: A program generation framework for fast kernels, in: IFIP Working Conference on Domain Specific Languages (DSL WC), Vol. 5658 of Lecture Notes in Computer Science, Springer, 2009, pp. 385–410.

[17] S. P. Jones (Ed.), Haskell 98 Language and Libraries: The Revised Report, Cambridge University Press, 2003. doi:DOI: 10.2277/0521826144.

[18] R. Bird, O. de Moor, Algebra of Programming, Series in Computer Science, Prentice-Hall International, 1997.

[19] P. Freyd, A. Scedrov, Categories, Allegories, Vol. 39 of Mathematical Library, North-Holland, 1990.

[20] S. MacLane, Categories for the Working Mathematician, Vol. 5 of Graduate Texts in Mathematics, Springer, 1998.

[21] H. Macedo, J. Oliveira, Matrices As Arrows! A Biproduct Approach to Typed Linear Algebra, in: Mathematics of Program Construction, Vol. 6120 of Lecture Notes in Computer Science, Springer, 2010, pp. 271–287.

[22] A. Bove, P. Dybjer, Dependent types at work, in: A. Bove, L. Barbosa, A. Pardo, J. Pinto (Eds.), Language Engineering and Rigorous Software Development, Vol. 5520 of Lecture Notes in Computer Science, Springer, 2009, pp. 57–99.

[23] R. B. J. T. Allenby, Linear Algebra, Elsevier, 1995.

[24] S. MacLane, G. Birkhoff, Algebra, AMS Chelsea, 1999.

[25] A. Tarski, S. Givant, A Formalization of Set Theory without Variables, AMS, 1987, AMS Col. Pub., volume 41, Providence, Rhode Island.

[26] M. F. Frias, Fork algebras in algebra, logic and computer science, Logic and Computer Science. World Scientific Publishing Co. (2002).

[27] M. Winter, A pseudo representation theorem for various categories of relations, Theory and Applications of Categories 7 (2) (2000) 23–37.

[28] S. Wolfram, et al., Mathematica: a system for doing mathematics by computer, Addison-Wesley, 1988.

[29] Y. Voronenko, Library generation for linear transforms, Ph.D. thesis, Electrical and Computer Engineering, Carnegie Mellon University (2008).

[30] R. S. Bird, Lectures on constructive functional programming, in: M. Broy (Ed.), Constructive Methods in Computer Science, Springer-Verlag, 1988, pp. 151–218.

[31] K. Goto, R. A. v. d. Geijn, Anatomy of high-performance matrix multiplication, ACM Trans. Math. Softw. 34 (3) (2008) 1–25.

[32] P. D'Alberto, A. Nicolau, Adaptive Strassen's matrix multiplication, in: Proceedings of the 21st annual international conference on Supercomputing, ICS '07, ACM, 2007, pp. 284–292.

[33] K. Abadir, J. Magnus, Matrix algebra. Econometric exercises 1, Cambridge University Press, 2005.

[34] H. Macedo, Matrices as arrows — Why categories of matrices matter, Ph.D. thesis, University of Minho, (In preparation) (2011).

[35] J. Magnus, H. Neudecker, The commutation matrix: Some properties and applications, The Annals of Statistics 7 (2) (1979) 381–394.

[36] A. Joyal, R. Street, The geometry of tensor calculus I, Advances in Mathematics 88 (1) (1991) 55–112.

[37] A. Joyal, R. Street, D. Verity, Traced monoidal categories, in: Mathematical Proceedings of the Cambridge Philosophical Society, Vol. 119, Cambridge Univ Press, 1996, pp. 447–468.

[38] K. Došen, Z. Petrić, Symmetric self-adjunctions and matrices, preprint available from http://arxiv.org/abs/math/0510039, last revision: 2011 (2005).

[39] D. Padua (Ed.), Encyclopedia of Parallel Computing, Springer, 2011, entry: *Spiral*, by M. Püschel, F. Franchetti and Y. Voronenko.

[40] J. N. Oliveira, Transforming data by calculation, in: R. Lämmel, J. Visser, J. Saraiva (Eds.), Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007. Revised Papers, Vol. 5235 of Lecture Notes in Computer Science, Springer, 2008, pp. 134–195.

[41] K. Došen, Z. Petrić, Self-adjunctions and matrices, Journal of Pure and Applied Algebra 184 (1) (2003) 7–39.

[42] W. E. Roth, On direct product matrices, Bulletin of the American Mathematical Society 40 (1934) 461–468.

[43] T. Altenkirch, J. Chapman, T. Uustalu, Monads need not be endofunctors, in: C.-H. L. Ong (Ed.), Foundations of Software Science and Computational Structures, Vol. 6014 of Lecture Notes in Computer Science, Springer, 2010, pp. 297–311.

[44] J. Jeuring, The derivation of hierarchies of algorithms on matrices, in: B. Möller (Ed.), Constructing Programs from Specifications, North-Holland, 1991, pp. 9–32.

[45] D. C. Kozen, Automata and Computability, 1st Edition, Undergraduate Texts in Computer Science, Springer, 1997.

[46] J. R. Lewis, B. Martin, Cryptol: high assurance, retargetable crypto development and validation, in: Proceedings of the 2003 IEEE conference on Military communications - Volume II, IEEE Computer Society, 2003, pp. 820–825.

[47] A. Eddington, Relativity Theory of Electrons and Protons, Cambridge University Press, 1936.

[48] S. Liu, G. Trenkler, Hadamard, Khatri-Rao, Kronecker And Other Matrix Products, International Journal of Information And Systems Sciences 4 (1) (2008) 160–177.

Appendix A. Calculational proofs postponed from main text

Calculation of (14, 15). The derivation of these facts is based on the existence of additive inverses and can be found in [20]. Let us see that of (14) as example:

$$
\begin{array}{ll}
 & \pi_1 \cdot i_2 = 0 \\
\Leftrightarrow & \quad \{ \text{ additive inverses } \} \\
 & \pi_1 \cdot i_2 = \pi_1 \cdot i_2 - \pi_1 \cdot i_2 \\
\Leftrightarrow & \quad \{ \text{ additive inverses } \} \\
 & \pi_1 \cdot i_2 + \pi_1 \cdot i_2 = \pi_1 \cdot i_2 \\
\Leftrightarrow & \quad \{ \text{ (11, 12) ; bilinearity (9, 10) } \} \\
 & \pi_1 \cdot \left(i_1 \cdot \pi_1 + i_2 \cdot \pi_2\right) \cdot i_2 = \pi_1 \cdot i_2 \\
\Leftrightarrow & \quad \{ \text{ (13) } \} \\
 & \pi_1 \cdot id \cdot i_2 = \pi_1 \cdot i_2 \\
\Leftrightarrow & \quad \{ \text{ identity (4) } \} \\
 & \pi_1 \cdot i_2 = \pi_1 \cdot i_2
\end{array}
$$

The other case follows the same line of reasoning. When additive inverses are not ensured, as in the relation algebra case, biproducts enjoying orthogonal properties (14,15) are the ones built on top of disjoint unions in distributive allegories [19, 27].

Proof of Theorem 2. The calculation of (11) for biproduct (47) is immediate:

$$
\begin{array}{rcl}
\pi_1^C \cdot i_1^C & = & \left[\begin{array}{c|c} 1 & 0 \end{array}\right] \cdot \left[\begin{array}{c} 1 \\ \hline -C \end{array}\right] \\
 & = & \left[\begin{array}{c|c} id_m & 0 \end{array}\right] \cdot \left[\begin{array}{c} id_m \\ \hline -C \end{array}\right] \\
 & = & id_m
\end{array}
$$

The calculation of (12) is similar. That of (13) follows:

$$
\begin{array}{ll}
 & i_1^C \cdot \pi_1^C + i_2^C \cdot \pi_2^C \\
= & \quad \{ \text{ (47) } \} \\
 & \left[\begin{array}{c} 1 \\ \hline -C \end{array}\right] \cdot \left[\begin{array}{c|c} 1 & 0 \end{array}\right] + \left[\begin{array}{c} 0 \\ \hline 1 \end{array}\right] \cdot \left[\begin{array}{c|c} C & 1 \end{array}\right] \\
= & \quad \{ \text{ fusion laws (26, 27) twice } \} \\
 & \left[\begin{array}{c|c} 1 & 0 \\ \hline -C & 0 \end{array}\right] + \left[\begin{array}{c|c} 0 & 0 \\ \hline C & 1 \end{array}\right] \\
= & \quad \{ \text{ blocked addition (37) } \}
\end{array}
$$

$$
\begin{array}{ll}
 & \left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & 1 \end{array}\right] \\
= & \quad \{ \text{ standard biproduct reflection (18,19) } \} \\
 & id
\end{array}
$$

Proof of Theorem 3. Only the calculation of (13) is given below, those of (11) and (12) being similar and actually simpler. Dropping identity matrix subscripts and relying on composition binding tighter than $\otimes$ to save parentheses, we reason:

$$
\begin{array}{ll}
 & (i_1 \otimes id) \cdot (\pi_1 \otimes id) + (i_2 \otimes id) \cdot (\pi_2 \otimes id) \\
= & \quad \{ \text{ (54) twice ; (4) twice } \} \\
 & (i_1 \cdot \pi_1 \otimes id) + (i_2 \cdot \pi_2 \otimes id) \\
= & \quad \{ \text{ (57) } \} \\
 & (i_1 \cdot \pi_1 + i_2 \cdot \pi_2) \otimes id \\
= & \quad \{ \text{ (13) } \} \\
 & id \otimes id \\
= & \quad \{ \text{ (55) } \} \\
 & id
\end{array}
$$

Calculation of fusion law (60).

$$
\begin{array}{ll}
 & [\ A \mid B\ ] \otimes C \\
= & \quad \{ \text{ (16) } \} \\
 & (A \cdot \pi_1 + B \cdot \pi_2) \otimes C \\
= & \quad \{ \text{ (57) } \} \\
 & (A \cdot \pi_1 \otimes C) + (B \cdot \pi_2 \otimes C) \\
= & \quad \{ \ C = C \cdot id \text{ twice ; (54) twice } \} \\
 & (A \otimes C) \cdot (\pi_1 \otimes id) + (B \otimes C) \cdot (\pi_2 \otimes id) \\
= & \quad \{ \text{ (58) } \} \\
 & (A \otimes C) \cdot \pi_1 + (B \otimes C) \cdot \pi_2 \\
= & \quad \{ \text{ (16) } \} \\
 & [\ A \otimes C \mid B \otimes C\ ]
\end{array}
$$

The elegance of this calculation compares favourably with the telegram-like proof of a similar result in [33] ("Kronecker product of a partitioned matrix") carried out at index-level, using "dot-dot-dot" notation.

Calculation of (73).

$$
\begin{array}{ll}
 & \epsilon \otimes id \\
= & \quad \{\ (70)\ \} \\
 & \left[\ \pi_1 \mid \pi_2\ \right] \otimes id \\
= & \quad \{\ (60)\ \} \\
 & \left[\ \pi_1 \otimes id \mid \pi_2 \otimes id\ \right] \\
= & \quad \{\ (58) \text{ twice; } (70)\ \} \\
 & \epsilon
\end{array}
$$

Calculation of (81).

$$
\begin{array}{ll}
 & \mathbf{vec}\,(A + B) = \mathbf{vec}\,A + \mathbf{vec}\,B \\
\Leftrightarrow & \quad \{\ \text{universal property (67)}\ \} \\
 & A + B = \epsilon \cdot (id \otimes (\mathbf{vec}\,A + \mathbf{vec}\,B)) \\
\Leftrightarrow & \quad \{\ \text{Kronecker product (56)}\ \} \\
 & A + B = \epsilon \cdot (id \otimes \mathbf{vec}\,A + id \otimes \mathbf{vec}\,B) \\
\Leftrightarrow & \quad \{\ \text{composition is bilinear (9)}\ \} \\
 & A + B = \epsilon \cdot (id \otimes \mathbf{vec}\,A) + \epsilon \cdot (id \otimes \mathbf{vec}\,B) \\
\Leftrightarrow & \quad \{\ \text{cancellation law (68) twice}\ \} \\
 & A + B = A + B
\end{array}
$$

Calculation of (83). This follows by instantiating cancellation law (77), for $A := \mathbf{vec}\,(B \cdot C)$, knowing that **vec** and **unvec** are inverses:

$$
\begin{array}{ll}
 & \mathbf{vec}\,(B \cdot C) \\
= & \quad \{\ (77)\ \} \\
 & (id \otimes (B \cdot C)) \cdot \eta \\
= & \quad \{\ \text{identity (4)}\ \} \\
 & (id \cdot id) \otimes (B \cdot C) \cdot \eta \\
= & \quad \{\ \text{bifunctoriality (54) ; associativity (3)}\ \} \\
 & (id \otimes B) \cdot ((id \otimes C) \cdot \eta) \\
= & \quad \{\ \text{canceling (77)}\ \} \\
 & (id \otimes B) \cdot \mathbf{vec}\,C
\end{array}
$$

Calculation of (84). We reason, minding subscripts k, m and n:

$$
\begin{array}{ll}
 & \mathbf{vec}_m\,(C \cdot B) = (B^\circ \otimes id_n) \cdot \mathbf{vec}_k\, C \\
\Leftrightarrow & \quad \{\ \text{(78) twice}\ \} \\
 & (id_m \otimes (C \cdot B)) \cdot \eta_m = (B^\circ \otimes id_n) \cdot (id_k \otimes C) \cdot \eta_k \\
\Leftrightarrow & \quad \{\ \text{(54) twice; (4)}\ \} \\
 & (id_m \otimes C) \cdot (id_m \otimes B) \cdot \eta_m = (id_m \otimes C) \cdot (B^\circ \otimes id_k) \cdot \eta_k \\
\Leftarrow & \quad \{\ \text{Leibniz}\ \} \\
 & (id_m \otimes B) \cdot \eta_m = (B^\circ \otimes id_k) \cdot \eta_k \\
\Leftrightarrow & \quad \{\ \text{see (A.1) below}\ \} \\
 & true
\end{array}
$$

The calculation relies on the commutativity of diagram

$$
\begin{array}{ccc}
m & & m^2 \xleftarrow{\eta_m} 1 \\
{\scriptstyle B}\downarrow & & {\scriptstyle id_m \otimes B}\downarrow \quad \swarrow {\scriptstyle \mathbf{vec}_m\, B} \quad \downarrow {\scriptstyle \eta_k} \\
k & & m \times k \xleftarrow[B^\circ \otimes id_k]{} k^2
\end{array}
\qquad \text{(A.1)}
$$

whose proof amounts to justifying equation

$$
\mathbf{vec}_m\, B \quad = \quad (B^\circ \otimes id_k) \cdot \eta_k \qquad \text{(A.2)}
$$

for $m \xrightarrow{B} k$. Changing variable $A := B^\circ$,

$$
\mathbf{vec}_m\,(A^\circ) \quad = \quad (A \otimes id_k) \cdot \eta_k \qquad \text{(A.3)}
$$

we see that it means that, by swapping the terms of the Kronecker product in a vectorization of a matrix $k \xrightarrow{A} m$, we produce a row-major vectorization of A instead of column-major one. This is amply discussed in [13].

The calculation of (A.3) relies on known properties of the commutation matrix:

$$
\begin{array}{lll}
\mathbf{vec}_m\,(A^\circ) & = & K_{mk} \cdot \mathbf{vec}_k\, A \\
 & \Leftrightarrow & \quad \{\ \text{(78)}\ \} \\
 & & K_{mk} \cdot (id_k \otimes A) \cdot \eta_k \\
 & \Leftrightarrow & \quad \{\ \text{natural-}K\ \text{(94)}\ \} \\
 & & (A \otimes id_k) \cdot K_{kk} \cdot \eta_k \\
 & \Leftrightarrow & \quad \{\ \text{(95)}\ \} \\
 & & (A \otimes id_k) \cdot \eta_k
\end{array}
$$

Calculation of (86):

$$
\begin{array}{ll}
 & \mathbf{vec}_{k+k'} \left[\!\!\left[\; A \;\middle|\; B \; \right]\!\!\right] \\
= & \qquad \{ \text{ either def. (16) } \} \\
 & \mathbf{vec}_{k+k'} \left(A \cdot \pi_1 + B \cdot \pi_2 \right) \\
= & \qquad \{ \text{ linearity of } \mathbf{vec} \text{ (81) } \} \\
 & \mathbf{vec}_{k+k'} \left(A \cdot \pi_1 \right) + \mathbf{vec}_{k+k'} \left(B \cdot \pi_2 \right) \\
= & \qquad \{ \text{ } \mathbf{vec} \text{ of composition (83) } \} \\
 & \left(id_{k+k'} \otimes A \right) \cdot \left(\mathbf{vec}_{k+k'} \, \pi_1 \right) + \left(id_{k+k'} \otimes B \right) \cdot \left(\mathbf{vec}_{k+k'} \, \pi_2 \right) \\
= & \qquad \{ \text{ } \mathbf{vec} \text{ of composition (83) } (\pi_1 = id_k \cdot \pi_1) \text{ and } (\pi_2 = id_{k'} \cdot \pi_2) \text{ } \} \\
 & \left(id_{k+k'} \otimes A \right) \cdot \left((\pi_1^\circ \otimes id_k) \cdot \mathbf{vec}_k \, id \right) + \left(id_{k+k'} \otimes B \right) \cdot \left((\pi_2^\circ \otimes id_{k'}) \cdot \mathbf{vec}_{k'} \, id \right) \\
= & \qquad \{ \text{ duality (24) ; scaled injections (59) ; } \eta \text{ def. (79) } \} \\
 & \left(id_{k+k'} \otimes A \right) \cdot i_1 \cdot \eta_k + \left(id_{k+k'} \otimes B \right) \cdot i_2 \cdot \eta_{k'} \\
= & \qquad \{ \text{ functor bilinearity (57) } (id_{k+k'} = id_k \oplus id_{k'}) \text{ } \} \\
 & \left((id_k \otimes A) \oplus (id_{k'} \otimes A) \right) \cdot i_1 \cdot \eta_k + \left((id_k \otimes B) \oplus id_{k'} \otimes B) \right) \cdot i_2 \cdot \eta_{k'} \\
= & \qquad \{ \text{ naturality (66) } \} \\
 & i_1 \cdot \left(id_k \otimes A \right) \cdot \eta_k + i_2 \cdot \left(id_{k'} \otimes B \right) \cdot \eta_{k'} \\
= & \qquad \{ \text{ } \mathbf{vec} \text{ definition (78) } \} \\
 & i_1 \cdot \mathbf{vec}_k \, A + i_2 \cdot \mathbf{vec}_{k'} \, B \\
= & \qquad \{ \text{ split definition (17) } \} \\
 & \left[\!\!\left[\dfrac{\mathbf{vec}_k \, A}{\mathbf{vec}_{k'} \, B} \right]\!\!\right]
\end{array}
$$